University of Oslo
Department of Informatics

Routing and Job Allocation in High Performance Clusters

Master thesis
60 credits

Jakob Aleksander Libak

11th February 2008

Abstract

To achieve good performance in a High Performance Computing (HPC) environment we need to have in addition to a large amount of computing resources, a high performing network with high bandwidth and low latency interconnecting the computing nodes and storage nodes in the cluster, and a processor allocation algorithm that gives a good utilization of the computing resources and a high throughput of jobs.

The InfiniBand Architecture is a network technology that offers high throughput, low latency and support for multiple upper layer protocols. In the recent years, InfiniBand has increased its popularity in the HPC environment.

In this thesis we investigate both routing in InfiniBand networks and processor allocation.

In the first part of this thesis we studied, optimized and implemented the LAyered SHortest path (LASH) routing algorithm for the OpenFabrics InfiniBand software stack.

In the second part of this thesis we studied processor allocation strategies and introduced a new processor allocation strategy, the Spiral allocation strategy.

Preface

This thesis is the result of my work for the master degree in informatics at the Department of Informatics at the University of Oslo. The work was performed at Simula Research Laboratory and at the Department of Informatics.

Acknowledgments

I would like to thank my supervisors Tor Skeie at the Department of Informatics and Simula Research Laboratory, Thomas Sødring and Åshild Grønstad Solheim at Simula Research Laboratory for their guidance, valuable comments and ideas during my work on this thesis. Without them this work would not have been possible.

I would also like to thank my parents for all the support they have given me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An area of great importance in computer science is High Performance Computing (HPC). To simulate difficult mathematical problems or to solve other problems that are very computationally demanding and cannot be solved on a typical desktop computer, researchers need large parallel super computers that have high computational power. These machines typically consist of thousands of computational nodes and I/O nodes interconnected in a high speed network. The Top500 super computer site [7] keeps a list of the fastest super computers today, it updates twice every year, as of June 2007 the fastest super computer is the IBM BlueGene/L [13] which has a peak performance of 368 teraflops and consists of 65536 computational nodes.

Scientific computing is not the only use of high performance computing, even businesses use high performance computing capabilities to accomplish their mission critical needs. In this setting the system is not used to solve a specific mathematical problem or model, but to provide a continuous high performance capability for processing real time transactions [24].

But thousands of computing nodes are not enough to get high performance, one also needs a high speed network that interconnects the nodes with efficient routing algorithms that prevent deadlock and that may even manage failures in the network, that is link failures or switch failures or any other hardware failure. One also needs a smart processor allocation strategy that utilizes the processors in the best possible manner to achieve the highest possible throughput of jobs.

1.1 Problem statement

This master's thesis is divided into two parts, as motivated in Section 1.3. The first part is about studying, optimizing and implementing the LAyered

SHortest path (LASH) [37, 51] routing algorithm for the OpenFabrics Subnet Manager (OpenSM) for an existing network technology called InfiniBand [48]. The second part concerns processor allocation and the implementation of a new processor allocation strategy that takes I/O communication into considerations where the placement of I/O nodes is not in the regular manner.

This thesis deals with two different problems: Can LASH be implemented and work with InfiniBand and can we implement a processor allocation strategy to perform well with regard to I/O communication.

1.2 Goals and sub goals

Our goals and sub goals for this master's thesis are as follows:

1.0 Investigate and optimize LASH performance and implement LASH in OpenSM

1.1 Implement a LASH simulator and test LASH for different topologies

1.2 Adapt LASH to work with OpenSM and InfiniBand

2.0 Investigate and implement a new processor allocation strategy

2.0 Implement and test processor allocation strategies

2.1 Implement a new processor allocation strategy

2.2 Simulate I/O performance

1.3 Motivation

The motivation behind the first goal is related to OpenFabrics [3]. OpenFabrics is an open source project that implements the InfiniBand software stack for Linux. In the current version, OFED 1.3, there exists four routing algorithms, Up*/Down*, FatTree routing, a shortest path routing algorithm and now also the LASH routing algorithm. Our motivation for implementing the LASH routing algorithm in this environment is to bring routing scalability with efficient performance to InfiniBand, and to see the actual implementation of this so far theoretical routing algorithm in a real system that is actually used by the industry is a big advancement for both research and industry. Another motivation is to bring our technology to the HPC landscape, which is of greater impact to our research.

The motivation behind the second goal relates to improvements of methods for processor allocation. Processor allocation has been researched since the late 1980's and several algorithms exists. We want to see how some of these algorithms perform in comparison to our processor allocation algorithm with regard to throughput of jobs and I/O performance.

1.4 Methods

For system performance evaluation there exist three different evaluation techniques, measurement, simulation and analytical modeling. [28] gives us considerations of how to decide which technique to use. Measurements can be used when a system, or a similar system, exists, while analytical modeling and simulations can be used when measurements are not possible.

Since we neither have an InfiniBand network available to test our LASH implementation nor an high performance computing cluster to experiment with our processor allocation algorithms, we will use simulation tools to test our implementations since analytical modeling is to complex. For the study of our LASH routing algorithm we have implemented a LASH simulator which has the ability of simulating LASH in different network topologies. For our processor allocation studies we have developed a processor allocation simulator that can easily be extended to implement any mesh based processor allocation algorithm. We have also implemented simulation tools to simulate network traffic for both parts of our thesis.

Since we are only interested in the steady state performance in our simulations, this is the performance after the system has reach a stable state, the initial transient periods are not included in our computations.

> "Since observations gathered during the initial transient periods do not charecterise the steady state, a natural idea is to discard all such observations before further analysis. This requires an estimation of the initial transient period."[41]

The problem of finding the end of the transient periods is called transient removal. Jain [28] lists six methods for transient removal: Long runs, proper initialization, truncation, initial data deletion, moving average of independent replications, and batch means. For both our LASH studies and the processor allocation studies we use initial data deletion to remove the transient periods.

1.5 Thesis layout

The rest of this thesis is laid out as follows. In Chapter 2 we give an introduction to various topics concerning *interconnection networks*.

In Chapter 3 we give an introduction to the InfiniBand Architecture and OpenFabrics, while Chapter 4 discusses the various implementations of the LASH routing algorithm we have developed and the results of simulations of these algorithms. Chapter 5 ties up this thread with a discussion of the implementation of LASH in the OpenFabrics Subnet Manager.

In Chapter 6 we start the second thread and give an introduction to *processor allocation* and give a presentation of the six processor allocation algorithms that we used in our work. In Chapter 7 we discuss the implementation of the processor allocation simulator we have implemented for our work. Then in Chapter 8 we discuss the results of our simulations of the various processor allocation strategies. Then in Chapter 9 we discuss I/O communication and propose a new processor allocation strategy.

In Chapter 10 we conclude this thesis with a discussion of our work.

Chapter 2

Interconnection networks

Interconnection network technologies are used in a large number of different applications. These include backplane buses, processor and memory interconnects, local area networks, interconnection networks for multicomputers, clusters and wide area computer networks. Examples of some interconnection network technologies and their properties are presented in [44]. Some examples are PCI (Peripheral Component Interconnect) [11], Fibre Channel [12], Ethernet [2] and InfiniBand [48].

In a data center environment the interconnection network connects the CPU nodes, the storage nodes and the internet gateways [35]. In this environment an interconnection network can be regarded as a short distance network that has high demands with respect to bandwidth, delay and delivery. With delivery we mean that we have no packet loss in the network and that packets are delivered in order.

Duato et al. give us a good overview of the following topics in interconnection networks, topology, routing and deadlocks in [23].

2.1 Topology

In [23] Duato et al. proposed a classification scheme for interconnection networks. We can divide interconnection networks into shared-medium networks, direct networks, indirect networks and hybrid networks.

Table 2.1 shows a classification of the different interconnection network topologies.

Group	Topology
Shared-medium	Local area network Backplane bus
Direct networks	Mesh k-ary n-cube Trees Cube connected cycles
Indirect networks	Butterfly networks Crossbar Clos networks
Hybrid networks	Multiple backplane busses Hierarchical networks Hypermesh

Table 2.1: Classification of topologies

2.1.1 Shared-medium networks

Shared-medium networks [23] include local area networks and the backplane bus. This is the least complex interconnect structure. Here the transmission medium is shared between all devices and only one device is able to use the medium at a time. This topology does not scale for a large number of processors because the shared medium becomes a bottleneck.

2.1.2 Direct networks

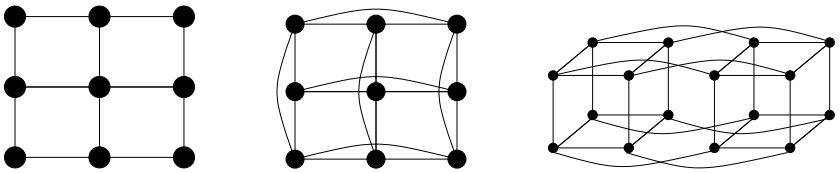
Direct networks or router-based networks [23] include strictly orthogonal topologies like mesh, torus and hypercube, and other topologies like trees and cube connected cycles. Figure 2.1 [23] illustrates three direct network topologies. In direct networks each node is directly connected to a subset of other nodes. This is a popular network architecture that scales for multiple processors in multiprocessor systems. The n-dimensional mesh, the k-ary n-cube also called torus, and the hypercube are the most popular direct networks.

An n-dimensional mesh has $k_0 \times k_1 \times \ldots \times k_{n-1}$ nodes, where k_i is the number of nodes along dimension i, where $k_i \geq 2$ and $0 \leq i \leq n-1$. Each node X is identified by n coordinates, $(x_{n-1}, x_{n-2}, \ldots, x_0)$ where $0 \leq x_i \leq k_i - 1$ for $0 \leq i \leq n-1$. Two nodes X and Y are neighbors iff $y_i = x_i$ for all i, $0 \leq i \leq n-1$, except for one j, where $y_j = x_j \pm 1$. Since the nodes have from n to $2n$ neighbors depending on their location, this topology is not regular.

In a k-ary n-cube, all nodes have the same number of neighbors. This topology is both regular and symmetric. It is different from the n-dimensional mesh that all k_i are equal to k and two nodes X and Y are neighbors iff $y_i = x_i$ for all i, $0 \leq i \leq n-1$, except for one j where $y_j = (x_j \pm 1) \bmod k$. The modular arithmetic adds wraparound channels to the k-ary n-cube.

A hypercube is also regular and symmetric. It is a special case of both the n-dimensional mesh and the k-ary n-cube. A hypercube is an n-dimensional mesh where $k_i = 2$ for $0 \leq i \leq n-1$, a 2-ary n-cube. It is also called a binary n-cube.

Other direct network topologies include cube-connected cycles, de Bruijn network and the star graph.



(a) 3×3 mesh (b) 3-ary 2-cube (torus) (c) 2-ary 4-cube (hypercube)

Figure 2.1: Examples of direct networks topologies

2.1.3 Indirect networks

Indirect networks or switch-based networks [23] include regular topologies like crossbar, multistage interconnection networks and irregular topologies. Multi-stage interconnection networks are divided into blocking and nonblocking networks. The butterfly network is an example of a blocking multistage network. An example of a nonblocking multistage network is the clos network. Figure 2.2 [21] shows a butterfly network. In indirect networks, the communication between nodes is carried through switches. Each node is connected to a switch. Regular topologies have regular connection patterns between switches while irregular topologies do not have any specific pattern.

A crossbar is a switching network with N inputs and M outputs allowing $\min(N, M)$ one-to-one interconnections without contention. Crossbar networks have been used in small-scale shared-memory multiprocessors.

Multistage interconnection networks connect inputs to outputs through a number of switchstages where each stage is a crossbar network.

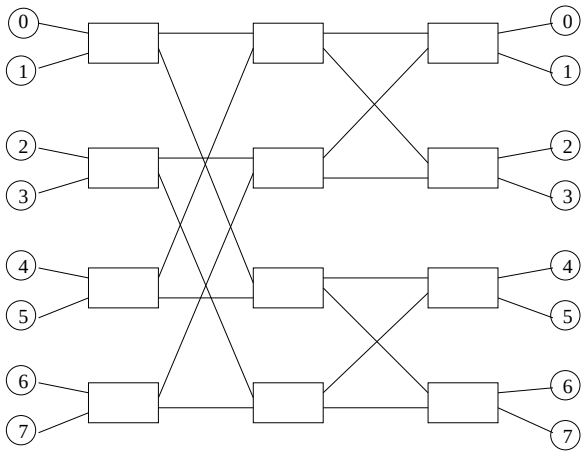


Figure 2.2: A butterfly network

2.1.4 Hybrid networks

Hybrid networks [23] include multiple-backplane buses, hierarchical networks and other hypergraph topologies like hyperbuses and hypermeshes. Hybrid networks combine mechanisms from shared-medium networks and direct or indirect networks.

Multiple-backplane buses are an enhancement of the backplane bus, the idea is to increase bandwidth by having multiple buses.

Hierarchical networks are a topology where different buses are interconnected by routers or bridges to transfer messages from one side of the network to the other side of the network. This technology is used in bridged LANs.

A hypermesh is a regular topology where nodes are arranged in several dimensions and each node is connected to all the nodes in each dimension through a bus.

2.2 Message switching and flow control

The interconnection network can be divided into three distinguished layers, the physical layer, the switching layer and the routing layer [23]. The physical layer includes the link level protocols for message transmission over physical channels between adjacent routers. The switching layer uses the physical layer to forward messages through the network. The routing layer establishes paths through the network.

This section focuses on the switching layer. With switching, we determine how internal switches in routers connect inputs to outputs and when messages are transferred over these paths. Flow control is used for synchronized

transfer of messages between and through routers in the network. We will discuss a few message switching techniques.

2.2.1 Circuit switching

In circuit switching [23], a physical path from the source to the destination is reserved before the data is transmitted. The path is reserved by injecting a routing probe into the network, this probe reserves the physical links as it travels through the network. When the probe reaches the destination, the complete path is reserved. An acknowledgement is sent back to the source.

The complete message is sent after the path has been reserved.

2.2.2 Packet switching

In packet switching [23], the message can be partitioned and sent as fixed-sized packets, where the first few bytes of a packet is called the packet header and contain routing and control information. Each packet is routed individually through the network from the source to the destination.

While the packet is transmitted through the network it is completely buffered at each node before it is transmitted to the next node. This method is also called *store and forward* switching. The packet header is read at each node and the routing information determines on which output port the packet is to be forwarded.

2.2.3 Virtual cut-through switching

Virtual cut-through switching [23] assumes a maximum packet size, and rather than waiting for the whole packet to be received, the packet header is read right after it has been received. The packet header and the following data bytes are then forwarded as soon routing decisions are taken and the output buffer is free. The message can cut-through to the input of the next switch before the whole packet has been received. We say that the packet is pipelined through the network. It is assumed that cut through routing occur at flit level where the routing information is contained in one flit. A flit is the unit of message flow control. The unit of message flow control in virtual cut through switching is one packet.

If the packet header is blocked on a busy output port, the whole packet has to be buffered.

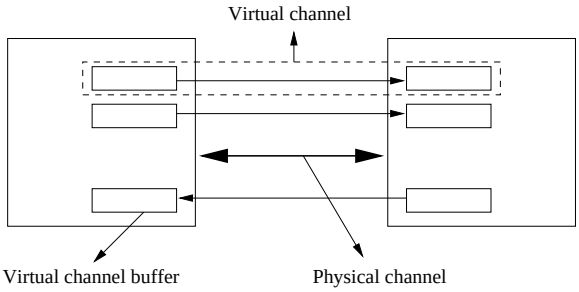


Figure 2.3: Virtual channels

2.2.4 Wormhole switching

Wormhole switching [23] is similar to virtual cut-through switching in the way that packets are pipelined through the network, but packets are divided into flits. Each output and input buffer in the network is large enough to store a few flits. The message is pipelined through the network at flit level. Wormhole switching does not assume a maximum packet size.

If a message is blocked it is blocked as is, and may occupy buffers in several routers in the network.

2.2.5 Virtual channels

Circuit switching, packet switching, virtual cut-through switching and wormhole switching assume that messages are buffered at the input or output of the physical channels, if a message occupies a channel buffer, no other message can use that physical channel.

A physical channel can support what we call *virtual channels*. A virtual channel is created by using separate input and output buffers for each virtual channel that share a physical channel. The virtual channels share a physical channel, but traffic in one virtual channel do not interfere with traffic in other virtual channels. Figure 2.3 [23] displays a model of virtual channels.

2.3 Routing

Routing algorithms are used to establish the path followed by each packet. There exists several routing algorithms for interconnection networks, all varying in how routing decisions are made. In [23] a taxonomy of routing algorithms is presented. In general we can distinguish between deterministic routing algorithms and adaptive routing algorithms.

2.3.1 Deterministic routing

Deterministic routing algorithms specify a path that is determined solely by the destination address, always providing the same path between all pair of nodes. The most popular deterministic routing protocols are the simplest ones. The simplest algorithm is called dimension-order routing. In this algorithm, packets are routed by crossing dimensions in strict order, routing in one dimension before routing in the next one. This algorithm is used in topologies that can be decomposed into several orthogonal dimensions. For n-dimensional meshes and hypercubes, this algorithm is deadlock free. XY routing is a dimension-order routing algorithm for routing in 2-D meshes.

2.3.2 Adaptive routing

Adaptive routing algorithms make routing decisions based on the network state. These algorithms allow more flexible routing, such as following alternate paths during congestion and faults. Adaptive algorithms are divided into partially adaptive and fully adaptive algorithms.

Partially adaptive algorithms use a trade-off between flexibility and complexity. Planar-adaptive routing [17] and turn model [25] are examples of partially adaptive routing algorithms.

Fully adaptive algorithms are able to select a path from all minimal paths in the network [23].

2.3.3 Routing in irregular topologies

Routing in irregular topologies is more complex than in regular topologies. In [53], Theiss discusses several routing algorithms for irregular topologies. Examples of these algorithms include Up*/Down* routing, MRoots, L-turn, Eulerian trail, fully adaptive routing with escape paths, shortest path routing with packet ejection/reinjection and the LASH routing algorithm.

In Up*/Down* routing [46], Up directions are assigned to links so that no cycles of Up links can be found. Down directions are assigned the opposite way. Packets are routed in the following manner, first packets travel zero or more Up links, then zero or more Down links. Transitioning from a Down link to an Up link is not allowed, this prevents cycles from forming and hence avoids deadlock. One way of assigning directions is to create a spanning tree where Up directions are assigned towards the root. For the links that are not part of the spanning tree, Up directions are assigned towards the node with the smallest spanning tree depth. If there is a tie, Up direction can

be assigned toward the lower Id node. The spanning tree can be found by conducting either a breadth first search [46] or a depth first search [45].

MRoots [36] is a variant of Up*/Down* which allows multiple roots. It reduces the hot spots around the single root in Up*/Down*. MRoots is a layered technique, for each network layer an independent Up*/Down* graph is calculated.

L-turn [30] is an adaptive routing algorithm for irregular networks that adds two more directions to Up and Down, Left and Right, where only one type of turn is allowed.

Eulerian trail [43] is based on an Eulerian graph. It routes adaptively between two acyclic unidirectional trails that contains all the channels in the network.

Fully adaptive routing with escape paths [49] requires virtual channels and it splits the layers between adaptive layers and escape layers. The escape layers use a deadlock free routing algorithm that consist of typically one layer. The adaptive layers use adaptive routing schemes. All packets are injected in the adaptive layers and if a packet encounters only occupied adaptive out channels at a node, it enters the escape layer avoiding deadlock.

In shortest path routing with packet ejection/reinjection packets always use the shortest path, but intermediate nodes can completely eject and later reinject messages.

LASH routing or LAyered SHortest Path routing is another scheme proposed and discussed in [37, 51, 53]. The idea is to divide the set of shortest paths for all source-destination pairs into subsets in a way that channel dependencies do not form any cycles in each subset. One layer is assigned to a subset of the shortest paths. The routing function R is defined by two sub functions R_{phys} and R_{virt}, R_{phys} defines the shortest path for each source-destination pair. R_{virt} defines the layer of packets following the path that is specified by R_{phys}. Deadlocks are avoided by ensuring that the channel dependencies formed by the shortest path in one layer do not form any cycles in the layer's dependency graph. The term *granular unit* is used to describe a set of source-destination pairs. The shortest paths are not added to a layer one by one but a granular unit is assigned en bloc. A granular unit is always chosen in such a way that the dependencies do not form any cycles.

Several versions of LASH exist, such as MP-LASH, A-LASH and LASH-TOR[37, 50]. A-LASH is LASH routing with switch adaptability, MP-LASH is LASH with multiple shortest paths and source adaptability and LASH-TOR is a generic transition oriented routing algorithm.

2.4 Deadlock

An important aspect of routing in interconnection networks is to avoid deadlocks. Deadlocks occur when a set of *agents* (packets) cannot advance in the network because the *resources* (buffers) requested by them are full, and all the packets are waiting for another packet in the set to progress. In a deadlocked configuration, all packets involved are blocked forever. Agents and resources can be viewed as wait for and hold relations [21]. There exists a resource dependency from a resources R_i to a resource R_{i+1} if it is possible for an agent holding R_i to wait for R_{i+1}. We denote this as $R_i \succ R_{i+1}$. Resource (channel) dependencies can be illustrated as a channel dependency graph with edges and vertices. Each resource is illustrated as a vertex in the graph and a dependency from resource u to resource v is illustrated as an edge from u to v [21]. A deadlocked configuration can be seen as a cycle in the channel dependency graph. Figure 2.4 [53] illustrates a deadlocked configuration. Here A is trying to send to C but is waiting on B, B is sending to D but is waiting on C, C is sending to A but is waiting D, and D is sending to B but is waiting on A.

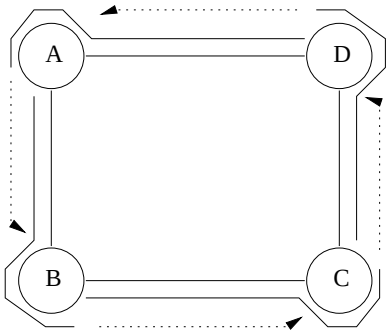


Figure 2.4: A deadlocked configuration

Figure 2.5 illustrates the channel dependency graph for the deadlocked configuration.

There are three strategies to deal with deadlocks, deadlock prevention, deadlock avoidance and deadlock recovery. Deadlock avoidance and deadlock prevention use different techniques to avoid deadlock completely while deadlock recovery uses techniques to resolve deadlock situations.

Deadlock avoidance grants resources to packets as the packet travels through the network. A resource is only granted if the global state is safe.

Deadlock prevention only grants resources to a packet in a way that requests never lead to deadlock. This can be done by reserving all necessary resources before transmission.

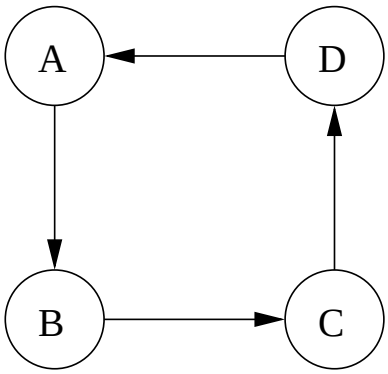


Figure 2.5: Channel dependency graph for Figure 2.4

Deadlock recovery uses techniques that have no restrictive routing functions but have mechanisms to detect and resolve deadlock situations.

2.5 Example systems

In this section we shall give three example of supercomputer systems, the IBM BlueGene/L, NEC Earth Simulator and the Cray T3E.

2.5.1 BlueGene/L

The IBM BlueGene/L [7, 13] is a massive supercomputer allowing speeds up to 360 teraflops, this means 360000 billions floating point operations per second. It is currently the leading supercomputer on the top500 list of supercomputers. It is used for many scientific simulations.

The BlueGene/L is a scalable system where the full system consists of 65536 computational nodes. Each node consist of one or more CPUs, memory and network interfaces. To achieve scaling in such a system, the system uses three different interconnection networks. The BlueGene/L is a cell based design where new building blocks can be added when more power is needed without causing bottlenecks.

The nodes are configured in a $32 \times 32 \times 64$ 3D torus, this means that every node is connected to 6 neighbors. The BlueGene/L also uses a global reduction tree to achieve global operations in milli seconds over all nodes. To achieve high throughput to disk, the full system consists of 1024 gigabit links to a global parallel file system.

The BlueGene/L uses a two phase partition allocation algorithm which must allocate a torus or mesh partition according to job requirements [13].

2.5.2 NEC Earth Simulator

The NEC Earth Simulator [9] was the worlds fastest supercomputer from 2002 to 2004, capable of 35.86 teraflops. It is used in earth sciences in for example global warming projections and solid earth interior dynamics research.

The system is a parallel vector supercomputer and consists of 640 processor nodes interconnected by single stage crossbar switches. Each processor node contains 8 vector processors, 16 GB of RAM, a remote access control unit and an I/O processor.

The interconnection network consists of 128 640×640 crossbar switches in addition to control units. Each switch has a bidirectional transfer rate of 12.3 GB/s. The total bandwidth in the network is 8 TB/s. The number of cables used in the interconnection network is $640 \times 130 = 83200$. The total cable length is 2400km.

2.5.3 Cray T3E

The Cray T3E [21] is a scalable supercomputer consisting of 272 nodes for a single cabinet machine scaling up to 8 cabinets for a full system which consists of 2176 computational nodes. The T3E's nodes is like BlueGene/L configured in a 3D torus network. The base configuration is a $8 \times 32 \times 8$ 3D torus. This only accounts for 2076 of the nodes, additional nodes for redundancy and operating systems are added on half of the Z dimension rings, bringing the total number of nodes to 2176.

The T3E uses cut-through switching in the network and every node has enough buffer space to store the largest packet. The routing function used in the T3E is dimension order routing.

Chapter 3

InfiniBand and OpenFabrics

The InfiBand Architecture [14, 48] (IBA) is a network technology that offers high throughput, up to 30Gb/s, low latency and support for multiple protocols. This technology is often used in a high performance computing (HPC) environment and according to the top500.org list of 06/2007 [7], 25,6% of the supercomputers use this technology. In 2005 this number was only 3,2%.

In this chapter we will first look into the InfiBand Architecture and describe the basics in the IBA specification. Then we look into OpenFabrics, an Open Source implementation of drivers and the subnet manager.

3.1 The InfiniBand Architecture

In the IBA specification [14, 48], a processor node is interfaced to the IBA fabric through one or more Host Channel Adapters (HCA). A HCA can have one or more ports. A port is a bi-directional interface which connects a device to a link. A link is a bi-directional connection between two ports on two different devices. A local network or a subnet is a set of ports and links with a common subnet ID and is managed by a common subnet manager (SM). The SM is responsible for discovering all devices in the subnet and configuring them at system startup. It also periodically sweeps the subnet to detect changes in the subnet topology.

An IO unit (IOU) is composed of a Target Channel Adapter (TCA) and one or more IO controllers (IOC). An example of an IOC is a mass storage array.

The IB Architecture provides direct access to the HCA, it bypasses the OS, this reduces the number of kernel context switches and memory operations. IB uses Remote Direct Memory Access (RDMA) to transfer data from the sender's memory to the receiver's memory without involving the

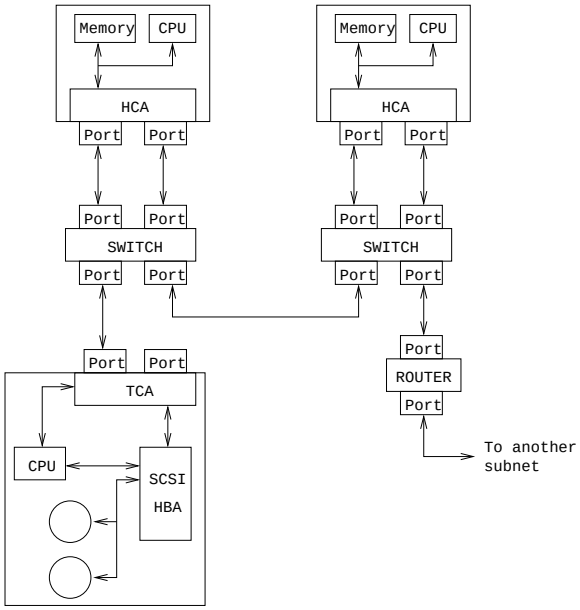


Figure 3.1: Example of an IBA system

CPUs. This eliminates the traditional system resource overhead associated with other network protocols.

IBA supports virtual lanes (VLs), creating several virtual links over one physical link. A VL is the same as a virtual channel that we described in Section 2.2.5. Each port implements two or more send and receive buffer pairs. The VLs are, according to the IBA specification, used for traffic management and Quality of Service (QoS). For example, some applications may fail if the messages are not delivered fast enough while other programs need not to deliver the messages that fast, in other words some programs have higher QoS demands than other programs. The IBA supports 2-16 VLs but the number of VLs implemented on a port is design specific. The number of VLs used for data has to be either one, two, four, eight or fifteen. VL0 to V14 are used for data packets while VL15 is used for subnet management.

Figure 3.1 [48] shows an example of an IBA system with two processor nodes, an IOU, two switches and a router.

3.1.1 Addressing of packets

A packet is used to send either a request or a response from one channel adapter (CA) to another CA. Each packet consists of a data payload field, one or two routing headers, a cyclic redundancy check (CRC) field and various headers. Two different routing headers exist in the specification, a local

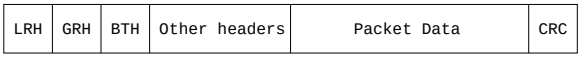


Figure 3.2: Packet format when using both LRH and GRH

routing header (LRH) and global routing header (GRH).

Every packet contains the LRH, as well as the following information, destination port local ID (DLID) and source port local ID (SLID). The port local ID (LID) is a 16-bit address that specifies a port in the IBA subnet. Every port in a subnet has a unique LID. The LIDs are distributed by the SM during system configuration.

The GRH is used when two CAs are not in the same subnet and it contains destination port global ID (DGID) and source port global ID (SGID). The port global ID (GID) is a 128-bit address that specifies a port globally. The upper 64 bits identify the subnet where the port is located and the lower 64 bits are the Globally unique ID (GUID) and uniquely identifies the port.

In addition to the LRH and possibly the GRH, a packet contains a Base Transport Header (BTH). This contains the following fields, an OpCode field identifies the type of a packet, a DestQP field identifies the destination QP within a CA. A QP is responsible for the handling of incoming packets. The Packet Sequence Number (PSN) field contains a 24-bit sequence number for a packet and is used to detect the loss of packets.

Figure 3.2 [48] displays the InfiniBand packet format.

3.1.2 Channel Adapters

The CAs are the real players in an InfiniBand network, they are the ones that send or receive information. Every CA has a specific number of ports and each port has a unique address. The CA is also called an end node and is defined as a device other than a switch, router or repeater. The definition of an end node according to the specification is:

> "An end node is any node that contains a Channel Adapter and thus it has multiple queue pairs and is permitted to establish connections, end to end context, and generate messages. Also referred to as Host Channel Adapter or Target Channel Adapter, two specific types of end nodes." [48]

3.1.3 Routers, switches and repeaters

If source and destination CAs are not directly connected, they are connected via switches and/or routers. That means the the first port a packet arrives

on, after output onto the first link, is on a switch or a router.

When a switch forwards packets within a subnet, it uses the DLID field to perform lookups in its forwarding table to determine onto which output port to forward the packet. The routing table for each switch in the network is generated by the subnet manager during startup or network sweep.

A router connects different subnets. The router uses the packet's DGID port address to determine in which subnet the destination CA is located. It uses the subnet ID field in the DGID address to perform a lookup in the routing table to determine onto which output port to transmit the packet. As with switches, the router's forwarding table is generated by the SM during startup or sweep.

A repeater is placed on a long link to compensate for a weakening signal strength.

3.1.4 The Subnet Manager

The subnet manager is a software entity that runs on a processor node in a subnet. Its role is to discover the topology of the subnet, that is discover all end nodes, switches and routers in the subnet, assign a common subnet ID to all ports in the subnet, assign an address to all ports in the subnet (the LIDS), establish the paths between all end nodes in the subnet and periodically sweep the subnet for topology changes. The topology changes when devices are added or removed.

A subnet must have at least one SM, but it can contain more SMs as a fallback mechanism. If it contains more than one, only one SM can be active at a time and this SM is referred to as the master SM while the others are referred to as standby SMs. During discovery, if an SM discovers another SM they negotiate to determine which one will be the master SM.

During normal operation the standby SMs periodically check if the master SM is still alive, if the master has died, the standby SMs negotiate to determine which one will replace the master SM.

3.1.5 Path query

When a CA wants to request information about routing between a source and destination port pair, a PathRecord Query is used. The results of the query is used to establish connections. The SM responds to the query with a PathRecord Query Result. A PathRecord includes among other things the DLID, SLID and Service Level (SL). The SL decides which VL that should be used. Each port that implements more than one data VL has to implement

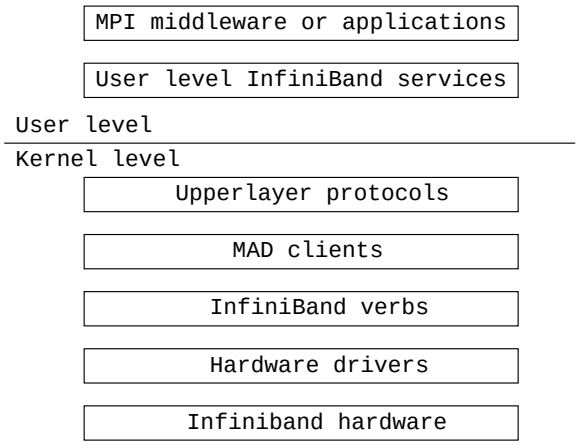


Figure 3.3: Overview of the Infiniband software stack

an SL to VL mapping table. The SM sets up these tables in each CA, switch and router during system configuration.

3.2 OpenFabrics

OpenFabrics is a hardware independent open source InfiniBand software stack for the Linux operating system created by the OpenFabrics alliance [27]. The InfiniBand software stack consists of both kernel-level and user-level components, Figure 3.3 [27] displays an overview of the InfiniBand software stack.

3.2.1 Kernel-level components

The kernel-level components consist of InfiniBand verbs which describe the action or function to take place, it is an API which sits on top of the hardware drivers, and the Management Datagram (MAD) services and agents which define the entry point for the upper layers to interact with the HCA. The upper layer protocols include IP-over IB, Socket Direct Protocol, SCSI RDMA protocol and Internet SCSI extension for RDMA.

3.2.2 User-level components

The user-level components consist of user-level APIs to allow access to the IB hardware, such as the Message Passing Interface (MPI).

3.2.3 OpenSM

OpenSM is the SM included with the OpenFabrics InfiniBand software stack. The SM runs in user space on top of the OpenFabrics stack. It is an open source implementation of the InfiniBand SM and the Subnet Administrator specified by the InfiniBand specification. Being an open source project, OpenSM is constantly under development.

The routing functions used by OpenSM is implemented as modules and which routing module that is used is specified when starting OpenSM.

3.3 Summary

In this chapter we have given a brief introduction to the InfiniBand specification and the OpenFabrics implementation of the InfiniBand stack. In Chapter 5 we describe the implementation of LASH for OpenSM.

In the next chapter we will discuss the LASH routing algorithm, the implementation of our LASH simulation tool and the simulation of the different versions LASH.

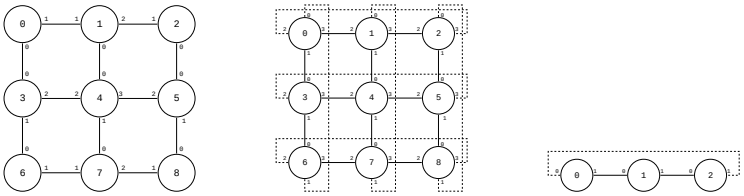
Chapter 4

LAyered SHortest path routing

The LASH routing algorithm was briefly discussed in chapter 3. But we will give a repetition of the algorithm.

The routing function R is defined by two sub functions, R_{phys} and R_{virt}. R_{phys} defines a shortest physical path between each source and destination pair. R_{virt} determines onto which virtual layer to forward a packet for every source and destination pair. The algorithm is as follows. First we find R_{phys} by calculating the shortest path between every source and destination pair. Then for every source and destination pair paths we try to assign a source and destination pair path to a virtual layer without closing any cycle of dependencies in that layer. If we cannot assign a path to a layer because we create a cycle of dependencies, we create a new virtual layer and assign the path to that layer. When this step completes we have found R_{virt}.

Before we implemented the LASH routing algorithm for the OpenFabrics SM, we implemented LASH in a simulator in written the C programming language, the same language OpenFabrics is implemented in, to test how LASH performs with different topologies of different sizes and to lessen the work of adding LASH to OpenSM. In the first section the initial implementation of the LASH routing algorithm in a LASH simulator is presented before we discuss the results of simulations for this implementation. In the second section we discuss the optimizations that were done to LASH to reduce the numbers of virtual lanes needed and give the results of these simulations. In the third section we will introduce a variant of LASH which we call LASH with return paths (LASH-RP). In the fourth section we discuss LASH and LASH-RP when we introduce link failures in the network. In the fifth section we give the results of packet simulations in a discrete event packet simulator called Conan for the various simulations done in the LASH simulator.



(a) Port numbering on a 3×3 mesh

(b) Port numbering on a 3×3 2D torus

(c) Port numbering on a ring with 3 nodes

Figure 4.1: Port numbering on various topologies

4.1 Original LASH simulator

The LASH simulator is implemented in the C programming language and uses the LASH routing algorithm to configure the forwarding tables in the switches in the network. The simulator can be divided into three parts. The first part is reading or generating the network topology and setting up the data structures, the second part is the LASH routing algorithm and the third part is the writing of the results to screen and routing and topology files to disk for later use in the packet simulator. The LASH routing algorithm code was ported to C from an existing C++ and Java code implementations.

4.1.1 Generation of network topologies

The network we want to simulate can be generated in the following two ways. The topology may be read from a topology file and all the necessary data structures are set up accordingly. The second way is to just specify what topology one wants, the current topologies supported are mesh, 2D torus and rings. If one generates a network topology this way, one just specifies the topology to simulate and the size of the network. This is specified in the command line arguments.

When generating the different topologies we keep the port numbering on the each switch the same. Figures 4.1 shows the port numbering on mesh, torus and ring respectively.

4.1.2 LASH data structures

Our LASH implementation uses four data structures: *switch*, *queue item*, *cdg vertex* and *reachable destination*.

The *switch* data structure implements a network switch and includes the following: a switch id, the number of active links a switch has, a routing table, a physical connection table which tells us which switch is connected on each port, an indication whether or not the switch is in the queue, this is used by our shortest path algorithm, and finally an indication if we have seen this switch before, this is also used by the shortest path algorithm.

The *queue item* implements an item in a queue and contains the following, a switch id which indicates which switch we have in the queue and a pointer to the next queue item. This is a null pointer if we are at the last element in the queue. The queue item is used by the shortest path search when calculating the shortest path between every source and destination pairs.

The *cdg vertex* implements a cdg vertex and is used by LASH to find cycles in the different VLs when paths established by the shortest path algorithm are added to a VL.

The *reachable destination* structure is used by LASH to generate the forwarding tables after the shortest path algorithm has established every shortest path in the network. It includes a switch id and a pointer to the next reachable destination structure.

In addition to these four data structure the LASH implementation uses global variables. Some of them include, a queue head which is a pointer to a queue item and is used by the shortest path search, a switch array which is an array of pointers to every switch, a cdg vertex matrix which is a three dimensional array of pointers to cdg vertices, the number of cycles indicates the number of cycles found in the network, and an adjacency matrix which tells us if two switches are neighbors, this is used by the shortest path algorithm.

4.1.3 LASH implementation

The LASH routing algorithm consists of three distinct functional stages.

1. The shortest path routing algorithm
2. The separation of source and destination pairs into layers
3. The balancing of layers

First, the shortest path algorithm used by LASH is an implementation of Dijkstra's shortest path algorithm using the adjacency matrix to find every neighbor and discover the network topology, then after the Dijkstra algorithm we generate the routing table in all switches from the paths discovered by this algorithm.

Second, when all routing tables in all switches are set up we want to separate all paths into different layers to avoid causing deadlock. The algorithm goes through all switches and tries for each source and destination pair to put this path in the first virtual layer, if a cycle is detected in this virtual layer, we try to put this path in the next virtual layer. To detect cycles in the network we use a Channel Dependency Graph (CDG) analysis.

Third, after the paths for all source and destination pairs have been grouped into a number of virtual layers, we try to balance the number of paths in each layer. We do not want the number of paths in the different virtual layers to be uneven. The balancing step takes a random path from the layer with the highest number of paths and tries to move it to the layer with the lowest number of paths. We do this until the number of paths in each virtual layer have been more or less balanced. Doing this step we use CDG analysis to make sure that we do not create cycles in any layer.

4.1.4 Outputting results

The last step of the simulator after we have created a network topology and simulated LASH for this topology is to output the results of the simulation. The result we are most interested in is the number of virtual lanes needed by LASH. In addition to the lanes needed we are interested in the simulation time, both total simulation time and the time used by each step in the LASH algorithm. For the total running time we do not include time used to generate the topology and writing of results.

After the simulation results are written we write to file both the topology and the routing tables, these files are later used for packet simulation in Conan, an event based network simulation tool, to simulate LASH performance. The topology file tells us how the network is connected and the routing file tells us how each switch should forward packets.

4.1.5 LASH simulations

We simulated LASH for the following topologies, ring, mesh and 2D torus of different sizes. We simulated rings with 1 to 18 nodes and mesh and 2D torus with sizes from 2×2 to 18×18.

Figure 4.2 presents the results for the number of virtual lanes needed by LASH for the different topologies. We see that for mesh we only need one layer no matter what the size of the mesh is, which is the same as dimension order routing. Actually, when analyzing the routing tables, we find that LASH reduces to dimension order routing for mesh. This is a result of the port numbering in the switches and the nature of the shortest path algorithm.

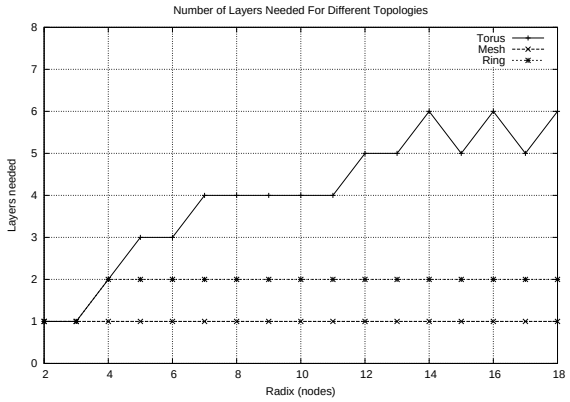


Figure 4.2: Virtual lanes needed for ring, torus and mesh

For rings we need only one virtual lane up to 3 nodes. But in rings with 4 nodes or more we need 2 virtual lanes. For torus the virtual lanes required grows, and the number varies, we see that for a 14×14 torus we need 6 virtual lanes while for a 15×15 torus we need 5 virtual lanes. It varies this way for the larger sizes. For a large torus network of even radix, such as 16×16, we need one layer more than for a torus of odd radix such as 17×17.

4.2 Optimization of LASH

We note from the previous section that LASH did not perform well for 2D tori with respect to virtual lane requirement. The number of layers needed for the larger sized networks grew and varied. Our goal with this optimization is to make LASH perform better in torus and other topologies. That is, to reduce the number of virtual lanes needed, decrease the path calculation time and reduce the memory usage.

It is important to reduce the number of virtual lanes required because the InfiniBand hardware that exist today do maximally support 8 VLs for data packets, and often only 4 VLs. Mellanox's InfiniHost HCA has two ports and supports up to 8 data VLs [10]. This is because the VLs are implemented as buffers in hardware and this has additional cost to the developers.

4.2.1 Changing the shortest path search

We started the optimization with implementing another shortest path search algorithm, the Breadth First Search (BFS). In the old LASH implementation the shortest path algorithm found every path in the network. The number

of paths quickly grows when the size of the network grows, which makes calculating the routing function inefficient and the memory usage will be large because a record of the paths is kept in all nodes. The new BFS based algorithm finds only one path between each source and destination pairs, this reduces both the time used for calculating the forwarding tables and the memory usage. Instead of using the adjacent matrix as we did in the old LASH implementation we use the output ports on each switch to find the switch's neighbors, which also reduces calculation times. In the old shortest path algorithm we had to go through the adjacent matrix n^2 times where n is the number of switches to find a switch's neighbors.

To make the BFS work we also needed to add a First in first out (FIFO) queue to store the visited switches. The queue used by the first search algorithm was a sorted queue but not FIFO. The FIFO queue makes sure that we always have shortest paths.

The old implementation of the shortest path algorithm was as follows:

```
void shortest_path(int ir, int num_switches) {
  int sw, dist, prev, i, channel;

  q_head = NULL;
  q_count = 0;

  enqueue(ir,0,NONE);

  while(q_count > 0) {
    dequeue(&sw, &dist, &prev);
    switches[sw]->mst_member = 1;

    if(prev != NONE) {
      channel = switches[prev]->used_channels;
      switches[prev]->dij_channels[channel] = switches[sw];
      switches[prev]->used_channels++;
    }
    for(i=0; i<num_switches; i++) {
      if(adj_matrix[sw][i]==1) {
        if(!switches[i]->mst_member) {
          if(switches[i]->q_member) {
            if(dist+1 == switches[i]->dist) {
              channel = switches[sw]->used_channels;
              switches[sw]->dij_channels[channel] = switches[i];
              switches[sw]->used_channels++;
```

```
            } else if(dist+1 < switches[i]->dist) {
              switches[i]->dist = dist+1;
              switches[i]->prev = sw;
            }
          } else {
            enqueue(i,dist+1,sw);
          }
        }
      }
    }
  }
}
```

This implementation is not optimal, for n switches we always have to go through the adjacent matrix n times to test if a switch is a neighbor. The enqueue function just puts a switch at the head of the queue and sets its distance and the previous switch. The dequeue function first finds the queue element with the shortest distance, then it goes through the queue from the queue head and removes the first element with distance equal to the minimum distance. It is basically a sorted LIFO queue.

The implementation of the new BFS based algorithm is as follows:

```
void breadth_first_search(int ir) {
  int sw, dist, prev, j, i, channel;

  q_head = NULL;
  q_last = NULL;
  q_count = 0;

  push(ir, 0, NONE);
  switches[ir]->visited = 1;

  while(q_count > 0) {
    pop(&sw, &dist, &prev);

    if(prev != NONE) {
      channel = switches[prev]->used_channels;
      switches[prev]->dij_channels[channel] = switches[sw];
      switches[prev]->used_channels++;
    }
```

```
    for(j=0; j<switches[sw]->num_connections; j++) {
      i = switches[sw]->phys_connections[j];
      if(!switches[i]->visited) {
        push(i, dist+1, sw);
        switches[i]->visited=1;
      }
    }
  }
}
```

As previously mentioned, instead of using the adjacent matrix the new BFS based algorithm uses the output ports on each switch to find the switches' neighbors. We also replaced the enqueue and dequeue functions with push and pop. The push method pushes a switch onto the queue and sets the distance and the previous switch, the pop method removes the switch at the head of the queue. The queue has been changed from a sorted LIFO queue to a FIFO queue. We see that we do not need to sort the queue when removing an element because the FIFO queue will always be sorted.

In the first implementation we also checked that if we found a neighbor switch we already had in the queue with a distance equal to the current distance we would add that switch to the current switch's channel. This means that we find every shortest paths in the network. The new BFS based algorithm does not test for this since we are only interested in one shortest path.

The channel array stores every next switch that are used by a path. This is used by the function that calculates the routing tables in each switch when the shortest path search completes.

4.2.2 Removing unnecessary data structures

When we had implemented the BFS we removed data structures no longer needed, among these was the adjacency matrix. We also removed some attributes in the switch data structure used by the old shortest path search.

4.2.3 Results from simulation

We ran the same tests as for the old LASH implementation. And they gave the following results.

We see in Figure 4.3 that for a mesh we only need one virtual layer, for a ring we need at maximum two virtual layers, these results are the same as for the original LASH algorithm. For a 2D torus, however, we need at

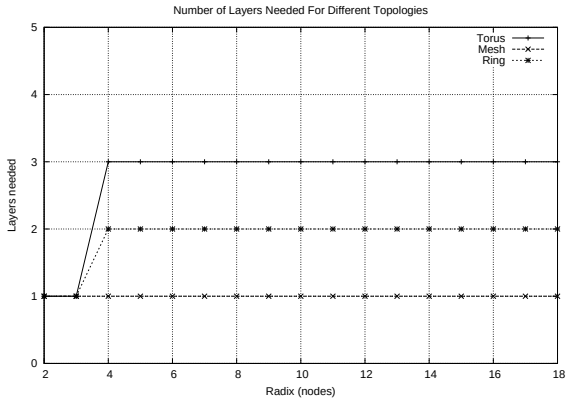


Figure 4.3: Virtual layers needed for various topologies with new LASH

maximum three virtual layers and this is a significant improvement from the old implementation.

We remember that many InfiniBand vendors do not deliver equipment that support more than 8 data VLs, and the previous version of LASH, would therefore not be scalable for large torus networks. The optimized version of the LASH routing algorithm, on the other hand would be scalable for networks of any size.

4.2.4 Comparison of the old and new implementation

We saw an improvement with regard to the number of virtual layers needed in the new LASH implementation with the BFS based shortest path algorithm. Figure 4.4 shows the running times in ms for the old and new LASH implementations and we see a big improvement in the running times. The experiment was done with 2D tori with sizes 2×2 to 18×18 and each experiment was repeated 3 times. The plot shows the average running times. We see that for the old LASH implementation the running times does not grow steadily like the new LASH implementation but varies for the different topology sizes.

Figure 4.5 is basically the same plot but the time used by the balancing step has been included for both new and old LASH, this experiment was only run once for each topology size, the network is still a 2D torus. We clearly see that the old shortest path algorithm uses more time than the new BFS based algorithm when the network size grows. While new BFS grows very slowly proportional to the network size, the old shortest path algorithm grows exponentially. We also see from this plot that the time used by the balancing step is related to the number of virtual layers needed.

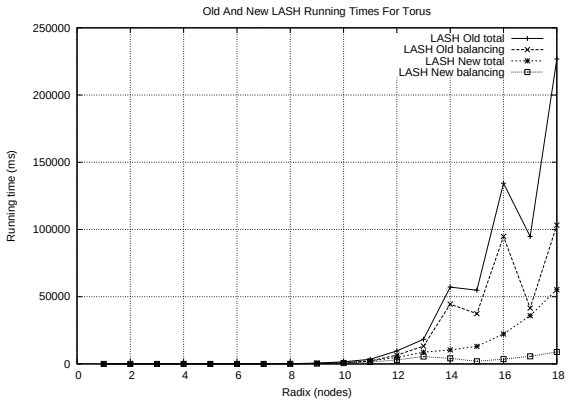


Figure 4.4: The running times for old and new LASH in 2D torus

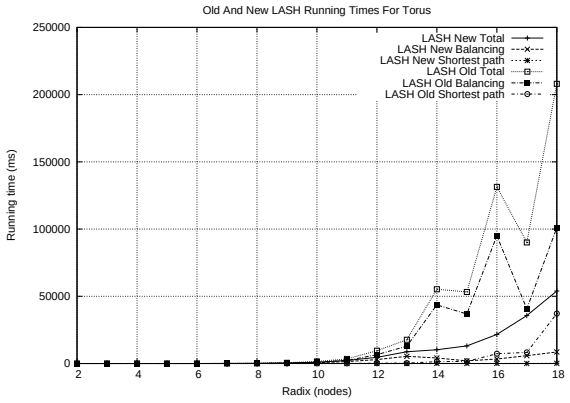


Figure 4.5: The running times for old and new LASH in 2D torus with balance time

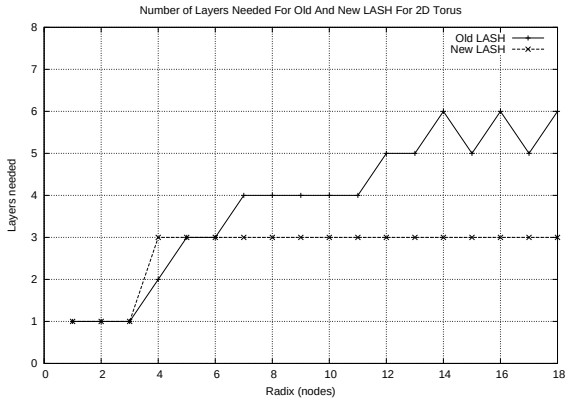


Figure 4.6: Comparison of old and new LASH implementation for 2D torus

The number of virtual lanes needed also improved with the new BFS based algorithm. Figure 4.6 shows the number of virtual layers needed for a 2D torus of size 1×1 to 18×18 for both old and new LASH. We see that for the new LASH implementation with the BFS based algorithm we need a maximum of 3 layers as the size of the torus grows.

When inspecting the forwarding tables for each switch we also observe an improvement, when generating a network where the port numbering is very strict, e.g. port 0 goes north, port 1 goes south, port 2 goes west and port 3 goes east, the BFS based algorithm results in routes that corresponds to the routes of Dimension Order Routing, in this case XY routing. The old shortest path search gave us routes only close to dimension order routing, the routes looked like both XY and YX routing was used.

Memory usage has also been improved, because we no longer need to store every shortest path in the network. Experimenting with LASH on a 18×18 2D torus on a machine with 4 gigabyte of memory, we found, using *top* that LASH with the old shortest path search used 1.1% memory while LASH with the BFS based algorithm used 0.7% memory. This means a 36.4% decrease in memory usage.

4.3 LASH with return paths

In addition to regular LASH routing we implemented a more strict version of LASH where the path from a source node a to a destination node b, and the return path from b to a must be in the same layer, we call this version of LASH for LASH with return paths, abbreviated LASH-RP. The motivation for creating LASH-RP is that higher level protocols used on top

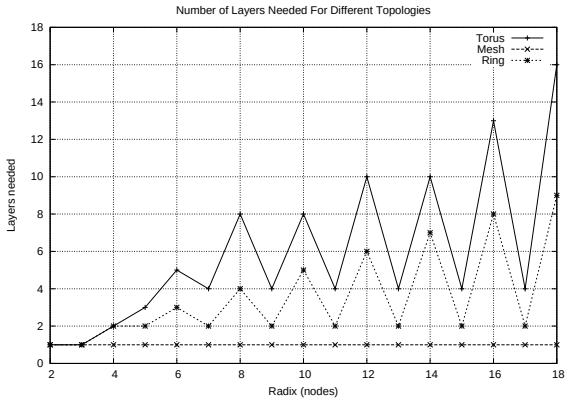


Figure 4.7: Virtual lanes needed for LASH-RP in different topologies

of the InfiniBand Architecture assumes that the path from a source node a to a destination node b and the return path from b to a are in the same layer. This is a problem that was reported to us by the InfiniBand Alliance during the the work with implementing LASH in the OpenSM.

We implemented LASH-RP with the new BFS and new data structures. The basic difference between LASH and LASH-RP is that when we try to place a path from a source node a to a destination node b in a certain virtual layer, we also try to place the return path from b to a in the same virtual layer at the same time. If we can put one path in a certain layer but not the return path we must try to put the path pairs in another layer. The same rules apply when doing the balancing step, we do not move one path to another layer without moving the return path.

4.3.1 Simulating LASH-RP

We did the same experiments for LASH-RP as we did with LASH. Figure 4.7 shows the number of virtual layers needed for mesh, 2D tori and rings of different sizes. When adding the constraint of having the return paths in the same virtual layer the number of virtual lanes needed increases for rings and tori topologies. For mesh networks we have the same results as before, only one virtual layer is needed.

4.3.2 Disadvantage with LASH-RP

We noted that for 2D tori and ring topologies we have an increase in the number of virtual lanes needed when the radix of the network is even.

We can see that for an even sized ring with n nodes we have

$$l = \frac{n}{2} \tag{4.1}$$

where l is the number of virtual lanes needed.

This disadvantage is a result of the nature of the BFS based search algorithm and the constraint of having the return path in the same virtual layer. When the topology is a ring of even size, the BFS search causes when going from a source node a to the destination node b with the longest distance from a, that we will always go left from both a to b and from b to a. This nearly causes a cycle in the network. When adding an additional such path pair to a virtual layer we will create a cycle in the channel dependency graph for that layer. We therefore need at least one layer for each such path pair, for n nodes there are $\frac{n}{2}$ such path pairs. Since the torus topology also contain rings, this problem also concerns tori.

4.3.3 Reduction of the need for layers

A possible solution to the issue caused by the return paths is to add dummy nodes in the network to achieve a radix of odd size, since we see that the problem only occurs when the radix is of even size. By increasing the size of each ring with one imaginary node, the increase in number of virtual layers required is avoided.

The dummy nodes are only used by the breadth first search when finding all paths in the network, and is not used by the LASH algorithm when separating paths into virtual lanes. The implementation of dummy nodes are done in the *switch* data structure, and just tells if there exists a dummy node between the current node and another neighbor. When we find a dummy node between the current node and a neighbor, we queue the neighbor with $dist + 2$ instead of $dist + 1$. This is to emulate a node between the two nodes. The BFS based algorithm has also been modified in such a way that we always keep a record of the shortest distance in the queue and if we encounter nodes with longer distances, these nodes are put back into the queue. This to ensure that the queue is sorted. The distance is the distance from a source node to the other nodes.

The breadth first search does not really know anything about the network topology, it only finds the shortest paths. We need additional logic that knows the network topology to handle the dummy nodes.

In the ring topology we only need to add one dummy node in the network, increasing its size with one. Where the dummy node is added is up to the added logic.

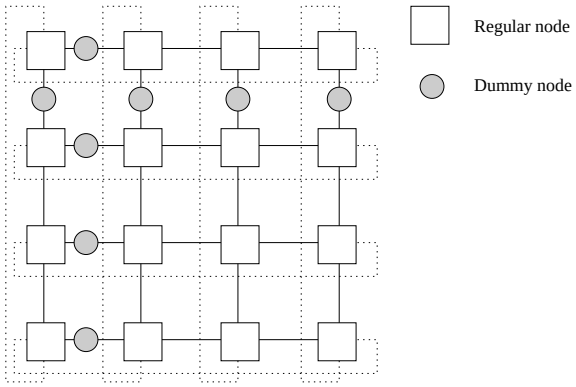


Figure 4.8: A possible placement of dummy nodes in a 2D torus

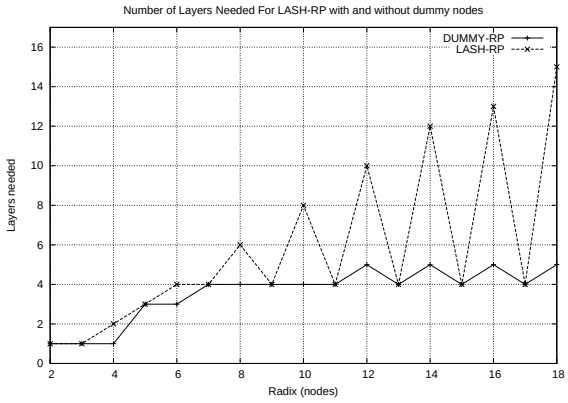


Figure 4.9: Virtual layers needed for LASH-RP with and without dummy nodes in a 2D torus

Figure 4.8 shows a possible placement of dummy nodes in a 4×4 2D torus. We have to put a dummy node in each ring in the torus.

4.3.4 Simulating LASH-RP with dummy nodes

Figure 4.9 shows the number of virtual layers needed for LASH-RP with and without dummy nodes in a 2D torus. We see that for a 2D torus with a radix of even size we have reduced the number of virtual layers needed to 5 when using dummy nodes. Without dummy nodes the number of virtual layers needed grows proportionally to the radix.

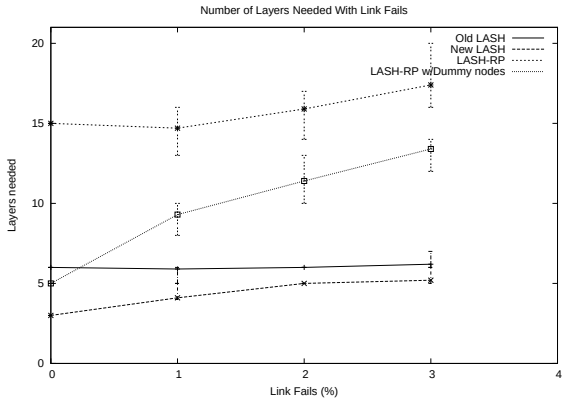


Figure 4.10: Virtual lanes needed when link failures are introduced

4.4 Link failures

We were also interested in how the implementation of LASH and LASH-RP performed with respect to layer requirements when we introduced link failures in the network. We know that over time links and switches fail, no network is fail proof, so we have to make sure that LASH and LASH-RP behaves well when failures occur.

We have simulated both the old and new LASH implementation in addition to LASH-RP with and without dummy nodes for an 18×18 2D torus networks and we let 1%, 2% and 3% of the network links fail. For a 2D torus with n nodes we have $2n$ links. Thus, an 18×18 2D torus with 324 switches have 648 links, this means that we simulate for 6, 12 and 19 link failures. All link failures are drawn randomly and to get a representative result we repeat the simulation 10 times, each with a different seed. The presented result is the mean of these 10 runs.

Figure 4.10 shows the results of the simulation, the lines displays the mean number of virtual layers needed and the vertical bars displays the minimum and maximum number of virtual layers needed at each failure level. We see that the original LASH algorithm need around 6 virtual lanes for all failure levels. For the LASH implementation with the new BFS based shortest path search algorithm, the number of virtual lanes needed is lower but grows slowly. It needs approximately 5 virtual lanes for 3% link failures. For LASH-RP with dummy nodes the number of virtual lanes needed increases very rapidly from 0 to 1% link failures, it nearly doubles. A reason for this is that the dummy nodes no longer have the intended effect. For LASH-RP without dummy nodes the number of virtual lanes needed is stable and around 16 virtual lanes are needed, it actually descends a little at 1% link

failures, but then increases slowly. The reason for this decrease is that some of the link failures break the loops in the torus network and thereby reduces the number of virtual layers needed.

We see from the plot that for LASH-RP with and without dummy nodes the number of VLs needed varies for a certain percentage of link failures. For regular LASH-RP we see that with 2% link failures the VLs needed varies between 13 and 18 with a mean of 15.4 for ten simulations. For 3% link failures the VL requirement varies between 14 and 17 with a mean of 15.8.

The number of VLs supported in the IBA specification is 16. We note that LASH-RP without dummy nodes requires 16 VLs in a 18×18 torus and with link failures it needs around 16 VLs and LASH-RP with dummy nodes needs 5 VLs without link failures and approximately 13 VLs with 3% link failures. To restrict the layer requirements in the presence of link failures, the LASH variations without the constraint regarding the return path seem to be the best choices.

4.5 Packet simulations with Conan

To study LASH and LASH-RP behavior with respect to packet routing in a network we used a packet simulator called Conan. Conan is an event based simulator based on the J-Sim framework [6]. We will discuss the J-Sim framework in Chapter 7.

4.5.1 Implementation of GenericRouting in Conan

To simulate the LASH routing algorithms we had to implement the routing function in Conan. The forwarding tables and topology file generated by the LASH and LASH-RP simulator were used as input to the simulator, these files were read by the new GenericRouting package we implemented. The GenericRouting package consists of four classes, *GenericSetup*, *GenericRouting*, *GenericPacket* and *GenericParameters*, and each class extends a Conan class.

GenericSetup is responsible for reading the parameters and setting up and starting the simulation.

GenericRouting is responsible for the routing. It starts by setting up the forwarding tables on each switch which is read from the forwarding table file. The forwarding table tells us which output port and VL to forward a packet onto to reach a certain destination. *GenericRouting* also implements the function to do the forwarding decision. This is done by reading the information in the packet header that tells us which destination end node on

which switch the packet is meant for. We only need these two functions. If a switch receives a packet with a destination end node connected to itself, it should simply forward the packet to the end node. If the destination end node is located on another switch, the current switch does a look up in its forwarding table to find onto which output port to forward the packet.

An important detail to note is that the LASH routing algorithm just finds the paths between the switches in the network. It does not take the end nodes into account.

GenericPacket contains packet information.

GenericParameters extends the ConanParameters class, and thereby inherits all common simulator parameters, and adds two new parameters, the file names of the forwarding table file and topology file.

4.5.2 Simulating LASH and LASH-RP

Before we could run the simulations in the Conan simulator we first generated the topology and routing files from the different LASH simulators. We generated LASH routing files for an 18×18 torus, that is a network of 324 switches. Every switch has two end nodes connected.

We simulated LASH with the old shortest path search, LASH with the new BFS based search, LASH-RP and LASH-RP with dummy nodes. Because Conan originally only supported 8 VLs and LASH-RP needed 15, we had to do some modification in Conan. This was done by overriding the existing implementation of a table for traffic class to which channel mapping table.

We ran the simulations with ten different network loads, from very low to very high. Each experiment was repeated five times, and the mean value is presented for each LASH algorithm. During simulations, Conan checks to see if the system has reached steady state, when it does, observations are being gathered. We ran the simulations for 30000 cycles. After this point, the gathering of observations are stopped, and we stop generating more packets. This removes the transient periods.

Figure 4.11 shows the result of the Conan simulations. On the x-axis is the network load in the total number of packet generated by the end nodes. On the y-axis is the number of delivered packets, that is the packets not lost due to overflow in the transmission buffers in the end nodes. During simulations end nodes discard packets if there is no space in the transmission queue. We see that for high loads many packets are lost for all variants of LASH, but LASH with the new BFS performs much better than LASH with the old shortest path algorithm which performs nearly the same as LASH-RP with dummy nodes. We see that LASH-RP without dummy nodes has the

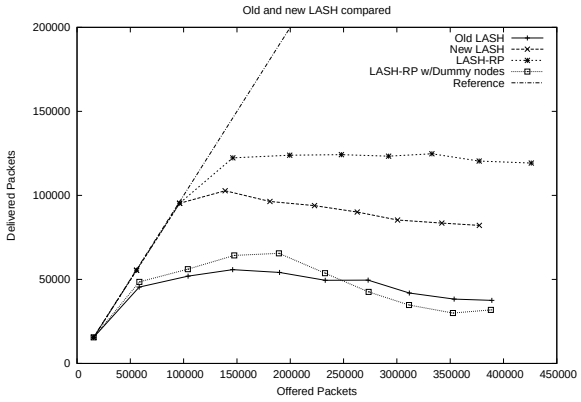


Figure 4.11: Throughput for the different LASH algorithms

best performance when the load is high. It has approximately 20% higher throughput of packets than LASH with BFS.

Since both LASH with BFS and LASH-RP use the same shortest path search their routes are the same but since LASH-RP has the constraint of putting the path between a source destination pair and the return path in the same layer, we know that the paths are distributed differently for the two LASH versions.

To test the theory that the distribution of paths across the set of virtual layers has a performance impact, we counted the number of times a link was used for all paths in each layer. The links were numbered and the results were printed to a file. Figure 4.12 shows the results for LASH with BFS. Along the x-axis is the link number and on the y-axis is the number of times a link was used by a path. We extracted the results from each separate virtual layer, and this plot shows the link usage for all layers plotted on top of each other. We see from the plot that the link usage varies significantly.

Figure 4.13 shows the result for LASH-RP without dummy nodes. In this plot we see that the link usage is concentrated at the bottom of the plot. Most links in each virtual layer is used less than 160 times, and few links are used more than 500 times. This even distribution of paths can play a significant role in the network performance, but at the cost of the need of many layers.

We simulated LASH with BFS one more time, but we modified the balance function to distribute the paths across 15 layers instead of 3. 15 is the same number of VLs used by LASH-RP. In Figure 4.14 the resulting throughput curve has been added to the plot of Figure 4.11. We see a small improvement from when only 3 VLs were used. We plotted the link usage in Figure 4.15. We see that many links are used less than 100 times, but still

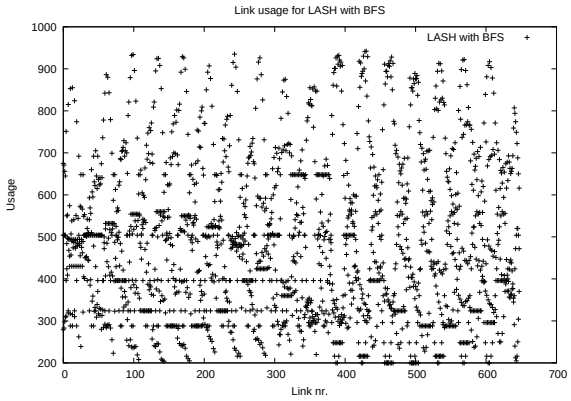


Figure 4.12: Link usage for LASH with BFS

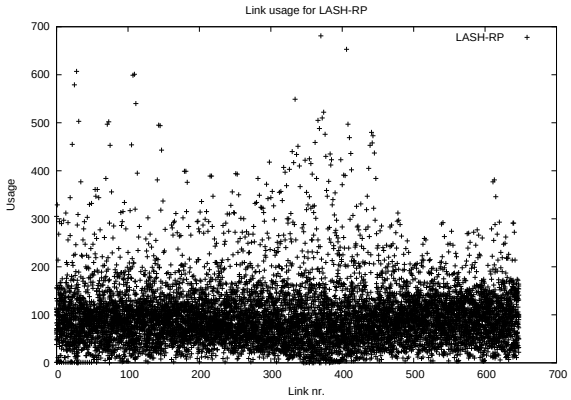


Figure 4.13: Link usage for LASH-RP

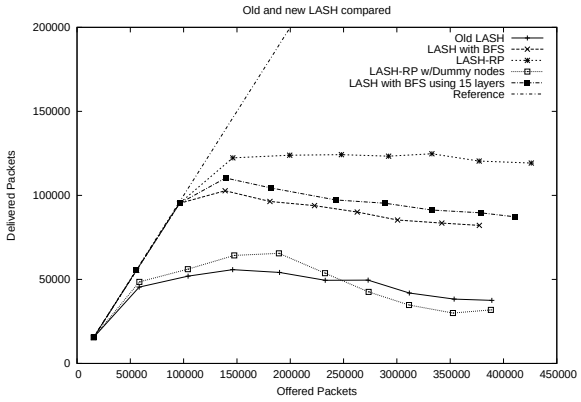


Figure 4.14: Result from simulations of LASH algorithms

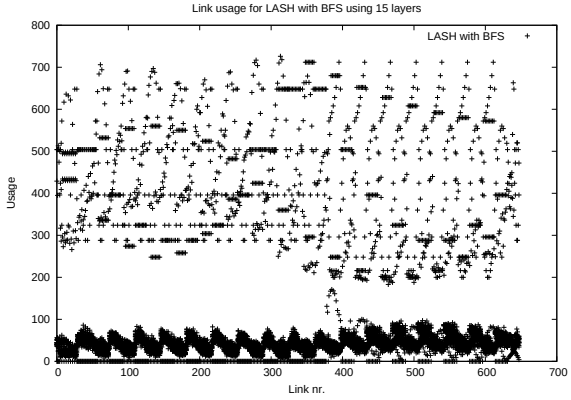


Figure 4.15: Link usage for LASH with BFS using 15 layers

many are used more than 300 times.

We see that when using dummy nodes the throughput decreased significantly. This is because the path calculation is different from regular LASH-RP, but with dummy nodes the number of virtual layers needed has also decreased. There is a trade off between performance and need for VLs.

4.6 Summary

In this chapter we have described how LASH was studied and optimized and we introduced LASH-RP. We saw that the new implementation of LASH performed much better than the old LASH implementation with respect to the layer requirement. We also demonstrated LASH-RP, that is, LASH with the constraint of having both the source-destination pair path and the the

return path in the same virtual layer. LASH-RP had issues for networks where nodes are connected in rings such as rings and tori. LASH-RP needs significantly higher number of virtual layers than regular LASH. The need for layers was reduced by introducing dummy nodes, pretending that each ring in the network has an odd number of nodes. From packet simulations we saw that LASH-RP performed significantly better with respect to packet throughput than the other implementations of LASH, but at the cost of extra layers needed. The new LASH achieved significantly higher throughput than the old implementation of LASH. LASH-RP with dummy nodes achieved throughput equal to the old LASH implementation.

In the next chapter we will discuss the implementation of LASH in OpenFabrics.

Chapter 5

Implementing LASH for OpenSM

Both the OpenFabrics Subnet Manager (OpenSM) and the LASH routing algorithm have been introduced and discussed in the previous chapters so we will not repeat them here. We have seen that since IBA uses VLs for traffic management and QoS it will be possible to use VLs also for routing algorithms that utilizes virtual layers to achieve deadlock freedom. LASH is one such routing algorithm.

Since the OpenSM routing functions is implemented as modules, we created the following file, *osm_ucast_lash.c*. This file contains both the data structures used and the LASH functions.

5.1 Adapting LASH for OpenSM

After having ported and tested the LASH routing algorithm, LASH was implemented into OpenSM. We kept the basic functions of LASH, the breadth first search algorithm, the separation of paths into different layers and the balancing algorithm, but we had to make some additions to LASH.

First, we had to read the network topology from the SM after it had swept the network to discover the network topology. This was given to the routing algorithm as a list of devices, and from this list the LASH data structures were built, that is the switches and their attributes. The data structures used were also modified to be dynamically allocated, as in the LASH simulator these were static.

A function to fill in the switches forwarding table was also created. It is not enough to calculate the routes without setting up the actual forwarding tables in the physical switches. The LASH data structures are only helper structures used for route calculation. Populating the forwarding tables is done after the LASH calculation is complete. The forwarding table is imple-

mented as an array where the index is the LID that is to be looked up and the value is the egress port on the current switch. What needs consideration is that the LASH routing algorithm only takes the switches into account when calculating the paths, the LIDs are not included in the algorithm. Therefore we need to keep a record of which switch each LID is connected to and on which port. For each switch in the network we go through each LID and find which switch it is connected to, if it is connected to the current switch, we just find the port that LID is connected to. If it connected to another switch we first have do a lookup into the LASH forwarding table to find which LASH egress port to use, from this we find the physical port to use. It is important to differ between LASH data structures and the physical structures used by the SM.

5.2 Selecting LASH as routing module

To be able to select LASH as the desired routing module, we also had to alter code in *osm_ opensm.c*. In this file there is a data structure called *routing_ engine_ module* that contains the name of the routing module and its setup function. Here we added the line.

```
{ "lash", osm_ucast_lash_setup }
```

With that LASH was available as a routing alternative within OpenSM. As of OFED 1.3, the current version of OFED, LASH is included as a routing function.

5.3 Selecting virtual layers

When a Channel Adapter (CA) wants to establish a connection with another CA, the initiator sends a PathRecord Query to the Subnet Manager (SM). The PathRecord typically contains DLID, SLID, Service Layer (SL) and other information that the CA needs to establish a connection. When the SM receives the PathRecord Query it does a query to its path database to find which DLID the CA should send to. The SM fills in the required information in the PathRecord and replies with a PathRecord Query result.

OpenSM is implemented in such a way that SLs are not used. In the future it could be used for traffic management or layered routing so the SL used in the PathRecord Query Result is always 0 or whatever the SL was in the PathRecord Query.

To have LASH work with OpenSM we have to have a DLID to SL mapping, we use a table for this purpose.

When the SM receives a PathRecord Query and the the SM fills in the SL it must test if LASH routing is used. OpenSM must still work for the other routing algorithms that have been implemented, and we do not want to use the DLID to SL mapping if another routing function is used. If LASH is used as the routing function we do a lookup in the DLID to SL table.

This means that to make LASH work with OpenSM, it is not sufficient to implement just the routing function but we also have to extend the OpenSM to fill in SLs in a PathRecord Query.

The modifications were done in the file *osm_sa_path_record.c* in the function *osm_pr_rcv_build_pr*. This is the function that builds the Path Record query result. In this function we added the following code:

```
if(strcmp(osm.routing_engine.name,"lash") == 0) {
  uint8_t sl = get_sl(src_lid_ho, dest_lid_ho);
  p_pr->sl = cl_hton16(sl);
} else
  p_pr->sl = cl_hton16(p_parms->sl);
```

First we test if the routing engine is LASH, then we do the lookup in our table to find which SL we should use and set the Path Record SL to the SL found. Otherwise, if LASH is not the routing engine, we use the SL already selected.

5.4 Testing the implementation

We did not have InfiniBand hardware to test our implementation, but OpenSM ships with a simulator that simulates various InfiniBand network topologies, one specifies the topology of switches and CAs in a given file. To have the OpenSM work with this simulator, the OpenSM has to be compiled in a specific manner with additional parameters.

A problem with this manner of testing is that we just see if the switches' routing tables are set up correctly. We cannot simulate data traffic and test whether or not our way of selecting VLs really work, or whether or not the network deadlocks.

In collaboration with industry partners, the original LASH code was tested in a real system and shown to work. We are currently working on an implementation to make LASH-RP also available in OpenSM.

5.5 Summary

With this chapter we end the first part of this thesis. In the previous chapter we discussed the implementation and optimization of LASH and LASH-RP. We saw how these performed, with respect to virtual layer requirement, throughput and link usage for ring, mesh and torus topologies of different sizes, with and without link failures. In this chapter we discussed the implementation of LASH in OpenFabrics, an open source implementation of the InifiniBand Network Architecture. We also saw how the selection of layers was accomplished. LASH is now a routing algorithm included in OpenSM with OFED 1.3. LASH is the first layered routing algorithm available for the HPC market.

In the next chapter we will leave the network layer and enter the second part of this thesis which is about processor allocation.

Chapter 6

Processor allocation

With this chapter we start the second part of this thesis. In this chapter we describe the research area known as processor allocation, in particular, we describe the existing algorithms which we have implemented and used for comparison with the method we propose.

As previously explained in Chapter 1, we chose simulations because we were not able to get access to hardware to test the processor allocation strategies. In addition simulations are better for scalability, we can run our simulations on various sizes of machines.

We assume a system model where a scheduler selects the next job from a job queue to run based on priority, the allocator then chooses which processors the job is placed on. The job then runs until it is complete and we do not allow preemption. This is called space sharing. Cplant [15, 16], a commodity based supercomputer at Sandia National laboratory [4] is an example of such a system.

A diagram of the flow in task scheduling and processor allocation is shown in Figure 6.1 [18]. A user program is run as several parallel tasks and each task requests a certain number of processors. After a task has been submitted to the multi processor system it is put into a waiting queue. The scheduler then selects the next task from the queue and tries to allocate the number of processors specified by that task. After a task has completed the processors that were used by that task are deallocated and a new task may be selected from the queue. If a task cannot be allocated because we are not able to meet that jobs requirements the task is put back into the waiting queue.

Many processor allocation strategies have been proposed. We can divide these strategies into two different categories, contiguous and non-contiguous processor allocation. Contiguous allocation strategies restrict allocated processors to be physically adjacent, while non-contiguous allocation strategies do not have this restriction.

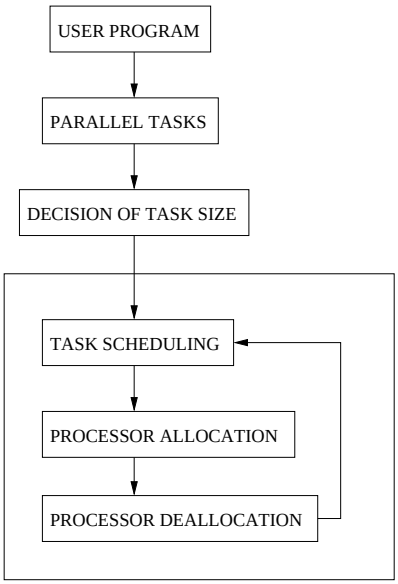


Figure 6.1: Scheduling and processor allocation

A problem that may occur in processor allocation is internal and external fragmentation. This results in reduced performance in a supercomputer environment. Internal fragmentation occurs when more processors are allocated to a job than the job requested. This typically occurs in architectures where the allocated area has to be of a certain shape, for example square submeshes of size 2^k for a $k \geq 0$. Figure 6.2 shows an example of internal fragmentation. The gray nodes are included in the allocated partition but not used because the job did only request 11 nodes. External fragmentation occurs when a sufficient number of processors are available to satisfy the request, but cannot be allocated because they do not satisfy the shape of the job requirement. This often occurs using contiguous allocation strategies where partitions have to be for example rectangular submeshes. Figure 6.3 shows an example where external fragmentation would occur if we tried to allocate a task which require a submesh with size 2×2. Non-contiguous processor allocation algorithms normally solves the problem of fragmentation since they do not require allocation of a contiguous area of a certain shape.

Table 6.1 shows a classification of some processor allocation strategies. These are described in the following sections.

The processor allocation strategies listed in Table 6.1 are mesh based processor allocation strategies. We must also mention that much research has also been done in processor allocation in torus networks [20, 29, 40, 42, 54], but our work concentrates on processor allocation in mesh network, so we will

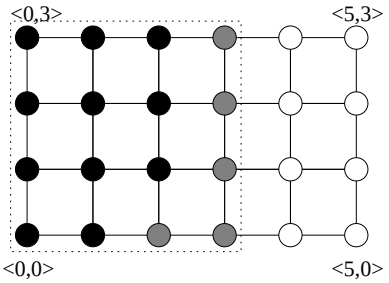


Figure 6.2: Example of internal fragmentation

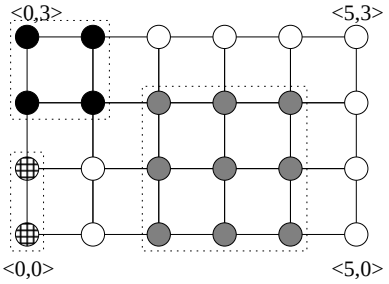


Figure 6.3: Example of external fragmentation

not discuss these algorithms in this thesis. The motivation for only studying mesh based processor allocation strategies is that most of the studies we could find, uses meshes as their topology of interest, and the traditional processor allocation strategies such as First Fit and Best Fit are mesh based. Mesh is also a widely used network topology in high performance clusters. Another consideration is Network On Chip, which uses the mesh topology. We could of course have chosen to study a different topology such as torus or multi stage networks which also are widely used topologies.

6.1 Contiguous processor allocation

For contiguous processor allocation algorithms, the allocated processors are restricted to be physically adjacent. In some algorithms they also need to form a subgraph of the original architecture [34]. This can be a submesh in meshes or subcube in hypercubes. In the following sections we will describe various processor allocation strategies for mesh connected systems.

The strategies we have used for comparison and implemented in our simulator model are First Fit, Best Fit and Adaptive Scan. We will describe these in detail in the following sections.

Scheme	Algorithm
Contiguous	2D-Buddy Adaptive scan Best-fit First-fit Frame sliding L-shaped Leap frog
Non-contiguous	Gen-Alg Manhattan median MC Multiple Buddy Paging Random

Table 6.1: Classification of traditional processor allocation strategies

6.1.1 First Fit

The First Fit and Best Fit strategies are described by Zhu in [56]. The First Fit algorithm allocates the first free submesh that are found of the size that a job requires.

Instead of scanning for a free submesh at each processor with a brute force strategy, the algorithm uses two arrays to decrease searching time. The value of each array element is either 0 or 1. The first array is called the *busy array* and tells us whether or not a processor is idle. The second array is called the *coverage array*, and is calculated for each incoming job. This array tells us whether or not a processor can be a base for an allocation. If the coverage array element is 0 the processor can be a base, if 1 the processor cannot be a base. The base of an allocation is the lower left corner node in the submesh.

Figure 6.4 shows how the coverages are calculated. The reject set is a set of processors that can not be used as a base for an allocation because the submesh would fall outside the mesh.

Each incoming task $T = (w', h')$ requires a submesh of width w' and height h'. First we scan each row in the busy array from right to left filling in the left coverages in the coverage array. Then we scan each column in the coverage matrix from left to right, each column is scanned from top to bottom filling in the bottom coverages in the coverage matrix. The first free processor we find in the second scan will be the base for the allocation, and all elements in the busy array corresponding to the new allocated submesh

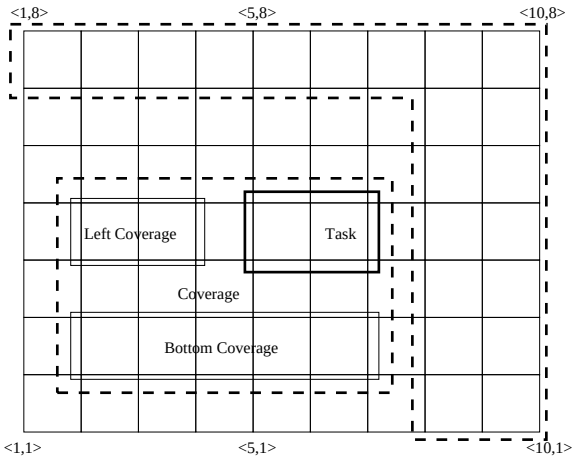


Figure 6.4: Calculating coverages in First Fit and Best Fit

will be set to 1.

For example, assume that we had the following busy array, showing an allocated 3×2 submesh in an 10×8 mesh topology:

$$B[10,8] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

If we had an incoming task $T = (4, 3)$, the coverage matrix would become the following:

$$C_T[7,6] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Remember that in First Fit we only scan the coverage matrix until we find the first free processor.

Deallocation is accomplished by setting the corresponding 1's in the busy array to 0.

6.1.2 Best Fit

The Best Fit strategy [56] is a variant of the First Fit strategy but instead of finding the first free area, it finds the best area that matches the job requirements, with the best area we mean the smallest area.

Each incoming task $T = (w', h')$ requires a submesh with width w' and height h'. Best fit calculates the coverage matrix the same way as the First Fit algorithm. After the calculation of the left and bottom coverages it scans the coverage matrix two more times. First we scan all rows, finding all consecutive sequences of 0's, marking the end elements as *left* and *right* respectively. Then we scan all columns, finding all consecutive sequences of 0's marking the end elements as *top* and *bottom* respectively. If an element is marked as *left* or *right* and *top* or *bottom* it is a corner, and will be added to a list of corners.

There are two heuristics to select a best-fit corner as a base for an allocation.

The first heuristic is to select the smallest area, that is the corner with the smallest area when the length of row and column sequences of 0's from that corner is multiplied.

The second heuristic is to select the corner with most busy neighbors.

In our implementation we use a combination of the two, we select the corner with the smallest area and in case of a tie between two possible nodes we prefer the node with most busy neighbors.

Deallocation is done the same way as in First Fit by setting the values in the busy array corresponding to the allocated submesh to 0.

6.1.3 Adaptive Scan

The Adaptive Scan strategy is described by Ding and Bhuyan in [22]. The idea is similar to the First Fit strategy discussed earlier, but instead of using a busy array, we use a busy set of all allocated submeshes. Also if we are unable to allocate the incoming task $T(w', h')$ we rotate the task and try to allocate $T(h', w')$ instead.

When an incoming task $T(w', h')$ arrives we first calculate the coverages for each job in the busy set with respect to T and add these to the coverage set. Then we scan each row in the mesh from the lower left corner node from left to right. If a node is not in the coverage set and not in the busy set we know that this is a free node that can be used as base for the allocation. We use this node as base for the allocation, and add the allocated submesh to the busy set.

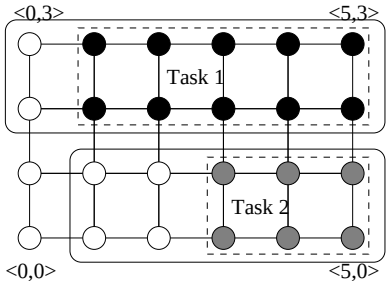


Figure 6.5: Coverages with respect to an incoming task $T(3, 1)$

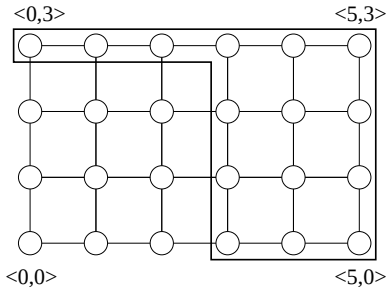


Figure 6.6: Reject set with respect to a task $T(4, 2)$ for a 6×4 mesh

Figure 6.5 [22] shows us the calculated coverages for two tasks in the busy set with respect to an incoming task $T(3, 1)$.

Figure 6.6 [22] shows the calculated reject set with respect to an incoming task $T(4, 2)$ for a 6×4 mesh.

6.1.4 Other contiguous strategies

There exists other contiguous processor allocation strategies for mesh connected systems and we will mention some of these briefly.

One early contiguous processor allocation strategy is the 2D Buddy strategy for mesh connected systems [32]. In this strategy all jobs are allocated to square submeshes of size $2^i \times 2^i$ and the mesh itself has to be square and of size $2^j \times 2^j$. This strategy suffers from both internal and external fragmentation.

Frame Sliding [19] is a strategy which is applicable to a mesh system of any size and any shape of submesh request, eliminating internal fragmentation. This strategy examines every candidate frame, until an available frame is found or all candidate frames have been checked. This strategy is not submesh recognition complete because the algorithm uses fixed strides.

The Leapfrog method [55] by Wu et al. uses a data structure called the R-array that represents the mesh, storing statistical information about the

occupied conditions of the mesh. This information can direct the allocation process to jump to the processors that can serve as a base of a free submesh.

FlexFold[26] is an allocation strategy which is similar to Adaptive Scan but in addition to try to allocate a submesh of either $a \times b$ or $b \times a$, it tries, if it is possible, to search for submeshes of size $\frac{a}{2} \times 2b$, $2a \times \frac{b}{2}$, $\frac{b}{2} \times 2a$ and $2b \times \frac{a}{2}$.

In [47] Seo discusses a fragmentation efficient node allocation algorithm for 2D mesh connected systems. The algorithm tries to allocate more flexible L-shaped submeshes to address the fragmentation problem that other contiguous algorithms are experiencing because they require square or rectangular submeshes. This strategy is called the LSA strategy, or the L-shaped submesh allocation strategy.

6.2 Non-contiguous processor allocation

Non-contiguous processor allocation algorithms deal with the problem of fragmentation. These algorithms remove the restrictions of contiguity of processors. Lo et al. discuss three non-contiguous processor allocation strategies, Random allocation, Paging allocation and the Multiple Buddy strategy [34].

Below we describe Random allocation, the MC allocation [39] and the Multiple Buddy strategy, we have implemented these non-contiguous strategies in our simulator for comparison for our processor allocation algorithm.

6.2.1 Random allocation

Random allocation is straightforward. In this strategy, if a task requests k processors, k processors are randomly selected from a list of available processors. If k processors are not available, the job is put in the waiting queue. In this scheme both internal and external fragmentation are eliminated since we always allocate an accurate amount of processors, and will always be able to allocate a job as long as a sufficient number of processors are available. A downside with this strategy is that the communication between processors nodes belonging to one job may pass through nodes that do not belong to the job, interfering with other jobs, and we can not guarantee that the processors are close to each other.

Figure 6.7 shows a random allocation of a task that requires 10 processors.

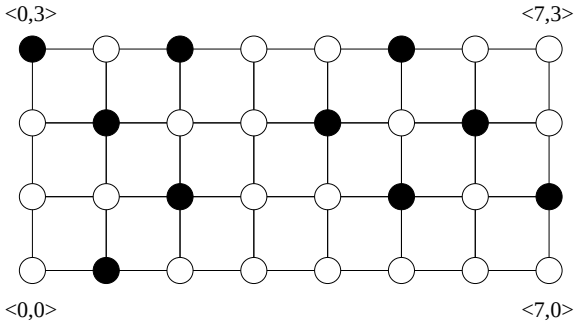


Figure 6.7: A random allocation of ten processors

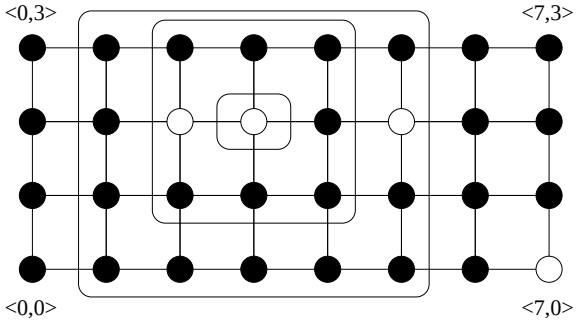


Figure 6.8: Shells centered around a free node using MC 1×1

6.2.2 MC

The MC [39] algorithm assumes that incoming jobs request processors in a particular shape, a submesh of size $w \times h$. Each free processor evaluates the quality of an allocation centered on itself. It counts the number of free processors within a submesh of the requested size centered on itself and within shells of processors outside the submesh. It then calculates the cost of an allocation based on the sum of the weight of each free processor. Then it repeats the previous steps after rotating the task size to $h * w$. The allocation with the lowest cost is chosen.

MC 1×1 [15, 16] is a special case of the MC algorithm. Here the shells are centered around a submesh of size 1×1, a single free node instead of a larger submesh.

Figure 6.8 shows the shells centered around a free node, using the MC 1×1 strategy where a job requires 3 processors. The cost of this allocation is 3, since the weighted cost of each processors in shell 1 is 1, in shell 2 the cost is 2 and so on. The cost of each processors inside the submesh is 0.

6.2.3 Multiple Buddy strategy

The Multiple Buddy strategy [34] is an extension of the 2D Buddy strategy and eliminates the fragmentation problem in 2D buddy. In the Multiple Buddy strategy several square blocks of size $2^i \times 2^i$ are allocated to a job. For 2D buddy, if a job requires five processors it has to allocate a block of size $2^2 \times 2^2$, resulting in internal fragmentation. In the Multiple Buddy strategy a block with size 2×2 and a block with size 1×1 can be allocated. External fragmentation is eliminated by breaking down the size of blocks, all blocks can be broken down to blocks with size 1×1.

The Multiple Buddy algorithm can be divided into three parts, system initialization, allocation and deallocation.

The system initialization is done at startup and only once. In this phase the mesh is partitioned into initial blocks. These blocks are non overlapping and are squares with side lengths of the power of two. A record of free blocks are kept in a *free block record* where *FBR[i]* keeps the free blocks of size $2^i \times 2^i$.

When a job comes into the system with a request for n processors, we first test if we have enough available processors. If this is not the case, the job is put in the waiting queue. We then factor the job, the factoring algorithm takes the job size as input and gives a request array as output. *Request_ Array[i]* keeps the number of $2^i \times 2^i$ blocks the job needs, this is the ith digit in the base 4 representation of the job size. After factoring, we try to allocate the correct number of blocks to the job, starting with the largest blocks. If we for the current block size do not have enough free blocks in the free block record, we try to generate buddies. This is done by partitioning a larger block into four smaller blocks. If this does not succeed because we do not have any larger free blocks available, we allocate the number of free blocks available in the free block record and instead increase the number of blocks needed in the request array for a smaller block size.

In the deallocation phase we free the blocks allocated by putting them back into the free block record. When freeing the blocks, buddies that can be merged are merged into larger blocks.

6.2.4 Other non-contiguous strategies

We will briefly describe other non-contiguous processor allocation strategies. These strategies have not implemented and include Paging, Gen-Alg and Manhattan median.

In Paging allocation [34], the entire mesh is divided into pages, this is square blocks of sides with length 2^s where s is page size. A request for k

processors is then satisfied by allocating free pages, until at least the number of processors requested have been allocated. An ordered list is used to keep track of unallocated pages. For $s = 0$ there will be neither internal or external fragmentation. For $k \geq 1$ internal fragmentation may occur, but we will have a higher degree of contiguity.

Bunde et al. [16] discuss in addition to Paging and MC a non-contiguous allocation strategies called Gen-Alg.

Gen-Alg [31] uses an algorithm to select a subset of k points from a set of n points to minimize their pairwise distance. This approximation even holds in non-mesh architectures. For each point p find the $k-1$ points closest to p and compute the total pairwise distance between all k points. Then return the set of k points with shortest pairwise distance.

In [15] Bender et al. discuss a non-contiguous processor allocation strategy called the Manhattan Median algorithm for 2D meshes. The algorithm is based on finding the smallest pairwise distance between a set of k points from a selected point p. This scheme has been implemented and tested in the Cplant facility. Manhattan Median algorithm is a variation of the Gen-Alg algorithm [15].

6.3 Using routing for better job allocation

To achieve better system utilization in a high performance supercomputer, e.g. a mesh connected system, it is possible to take advantage of topology agnostic routing to allocate contiguous areas with other shapes than submeshes.

The best possible utilization today is achieved with non-contiguous allocation strategies such as MC 1×1, but these allocations are not routing contained, which means that the traffic within a job may also affect other jobs. Submesh based processor allocation strategies are normally routing contained because in a mesh we usually use dimension order routing such as XY routing. These strategies have low system utilization, however, because of their strict requirement of allocating rectangular partitions, we see that external fragmentation occurs mainly for high system loads.

One way to achieve better system utilization is to use more flexible routing strategies than dimension order routing. For the UDFlex [52], processor allocation strategy, the underlying routing function is Up*/Down*, and instead of allocating submeshes, subgraphs are allocated.

6.3.1 Up*/Down* based processor allocation

The Up*/Down* based processor allocation strategy, UDFlex, allocates subgraphs instead of submeshes. Instead of assuming a mesh where the underlying routing algorithm is dimension order routing, the UDFlex algorithm assumes that the underlying network uses Up*/Down* routing. The UDFlex allocation strategy allocates Up*/Down* subgraphs from the Up*/Down* graph calculated by the routing algorithm. Therefore this allocation strategy, unlike the contiguous and non-contiguous strategies does not have to know anything about the underlying network topology, only the Up*/Down* graph given from the routing algorithm. The UDFlex algorithm may be used for any network topology including meshes.

For the mesh topology, a disadvantage with the UDFlex strategy is that Up*/Down* is not the optimal routing algorithm, dimension order routing is, in general, more efficient. Up*/Down* is for instance known to cause hot spot around the root node in the Up*/Down* graph.

6.4 Summary

In this chapter we have provided a literature review of the traditional processor allocation strategies. We explained their characteristics and where relevant provided examples. This chapter may also serve as a summary of the area of processor allocation for an interested reader.

In the next chapter we will discuss the implementation of the processor allocation simulator used for our work. This simulator tool was also used in [52].

Chapter 7

The processor allocation simulator

To be able to run the experiments discussed in the next chapter, a processor allocator simulator had to be implemented. This simulator was implemented in the Java [1] programming language using the J-Sim [6] framework.

We need to mention that Lo et al. have developed a processor allocation simulator with a graphical interface called ProcSimity[8] for their work which is available for download.

7.1 J-Sim

J-Sim is a component based, compositional simulation environment. The component based architecture consists of components, ports and contracts.

Figure 7.1 [6] displays the component based architecture of three components connected through ports.

The basic entity in the architecture is the component. The application can be viewed as a composition of components. The components are lightly coupled. They communicate with each other by connecting their ports together and they are bound to certain contracts.

The components communicates with each other through their ports. Each component can have several ports. The programming interface between a component and its ports is well defined. The component only interfaces with its own ports, meaning that components can be developed and implemented independently and integrated later in the development process.

The behavior of each component is specified by the port contract and the component contract. The port contract is bound to a specific port or group of ports, and specifies the communication pattern between the component

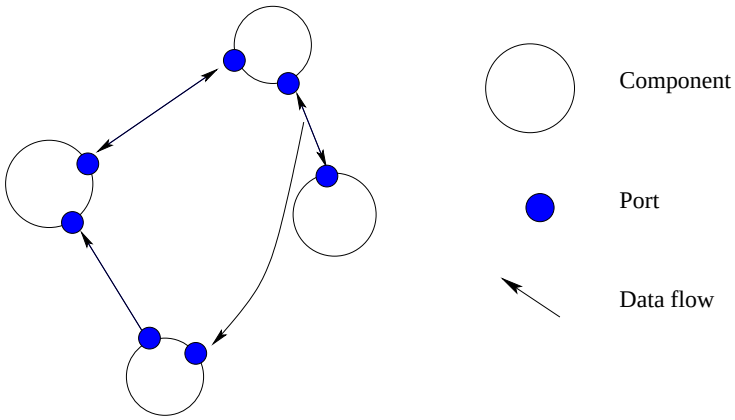


Figure 7.1: Component based architecture

owning the port and the component connected to the port. The component contract is a specification that characterizes the input and output relation of the component. It describes how a certain component reacts to data arriving at its ports. This can be how a component processes the data, updates data or generates output data on certain ports.

A contract specifies how a caller and a callee fulfill a function. It specifies the causality of exchange of information between components but not the components that participate in the exchange. Components acting as the caller and callee are bound during system integration to fulfill the contract.

A component can be reused in other applications with the same contract context.

An analogy to the component-based architecture is the integrated circuit architecture, where hardware modules are assembled by connecting a set of chips through their pins. When the signals arrive at the pins of a chip, the chip performs certain tasks, and may send signals to other pins. Figure 7.2 [6] displays the analogy between a J-Sim component and an IC chip.

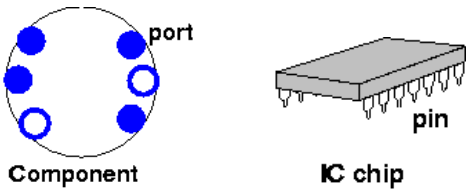


Figure 7.2: Analogy between a J-Sim component and an IC chip

7.2 Processor allocation simulator

The processor allocation simulator was implemented to study the processor allocation strategies discussed in the previous chapter, that is First Fit, Best Fit, Adaptive Scan, Random, MC 1×1, Multiple Buddy and the new irregular processor allocation strategy, UDFlex. Further in our studies we use First Fit, Best Fit, Adaptive Scan, Random, MC 1×1 and Multiple Buddy for comparison with our new processor allocation strategy.

The processor allocation simulator is composed of the following main components. A job generator, job scheduler and the processor allocator. We will discuss these parts in the following sections. Figure 7.3 shows a model of the processor allocation simulator. This is our model of the real world.

The simulator consists of three J-sim components, the job generator, the job scheduler and the job allocator. The job generator is connected to the job scheduler and and job scheduler is connected to the processor allocator. The processor allocator simulator acts on three kind of events, job generation, job start and job completion. When a job is generated, the simulator sleeps until the job is due to start. Then a wakeup is issued and a job is sent to the port and the scheduler puts the job in the queue for allocation. If the queue is empty, the job is immediately sent to the processor allocator which tries to allocate the job. If the allocation is successful, the job will run until completion. Otherwise, if the allocation was unsuccessful the job will be put back into the queue and will wait there until other jobs complete. When a job completes, it is deallocated and the corresponding processors are freed. If the queue is not empty at this time, we try to allocate the job at the head of the queue.

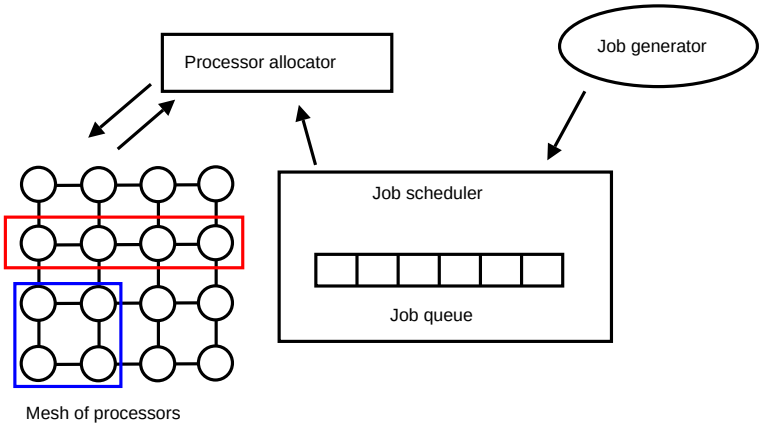


Figure 7.3: Model of the simulator

7.2.1 Job generator

The job generator is responsible for generating jobs. The job generator consists of three parts, a job manager, a job generator and a job creator.

The job manager is the main part and is also a J-Sim component with ports and contracts, and is responsible for generating jobs at the correct time and to send these to the job scheduler.

The job creator can be viewed as a user in a HPC environment. This is the part that creates the jobs, generating the job size and job duration. The original thought was that we could simulate several users, using several job creators. In our studies we use only one job creator for simplification.

The job generator implements methods to draw jobs from the certain distributions.

The jobs are of random size, running time and interarrival time, with interarrival time we mean the time between jobs generated by the system. The size are drawn from two uniform distributions. For both contiguous and non-contiguous job allocation strategies we draw the width w and height h of the job from two separate distributions. For non-contiguous strategies and the UDFlex strategy, the width w and height h is multiplied to get the number of processors needed for the job. The maximum and minimum width and height of a job is specified when starting the simulation.

The job running time is drawn from a uniform, exponential or Poisson distribution. The distribution used is specified when starting the simulation.

The interarrival time is decided in a similar manner as the job running time.

When using the uniform distribution min and max interval for both job duration and job interarrival time are specified. When using the exponential or Poisson distribution mean job duration and job interarrival time are specified.

The uniform distribution is generated using the Random class which exists in Java, while to generate the exponential or Poisson distributions we use a math library called *Commons-Math* [5]. This is a library containing lightweight components implementing mathematical and statistical functions which are not available in the Java language.

7.2.2 Job scheduler

The job scheduler is responsible for selecting a job from the queue and send it to the processor allocator.

The scheduler is a straight forward first come first serve scheduler meaning that the job at the head of the queue is allocated first. If a job cannot be

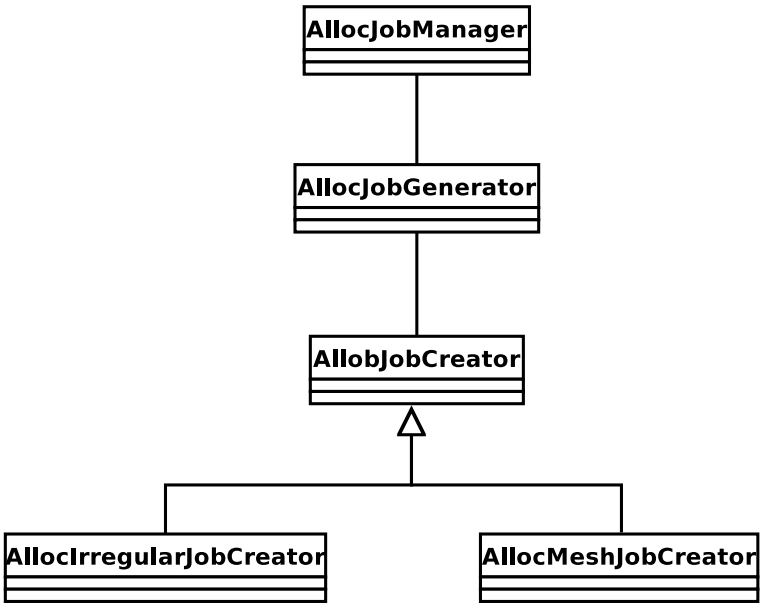


Figure 7.4: UML of the job generator

allocated it will stay in front of the queue until it can be allocated.

7.2.3 Processor allocator

The processor allocator is responsible for the allocation and deallocation of jobs. We have implemented the following processor allocation strategies, First Fit, Best Fit, Adaptive Scan, Random, MC 1×1, Multiple Buddy and our Spiral strategy.

When a job is sent to the allocator from the scheduler, the allocator uses the scheme specified when starting the simulation, trying to find a set of free processors specified by the incoming job's requirements.

The processor allocator is also responsible for the deallocation of jobs. When a job completes, an event is issued and the corresponding deallocation scheme is called. This routine frees the processors that the corresponding job was allocated to.

The processor allocator is also responsible for gathering data from the simulations, that is system utilization, system fragmentation, job service time and job queuing time.

7.2.4 Running simulations

To run simulations we first choose which processor allocation strategy we will use, e.g. First Fit. The processor allocation simulator is run from the command line with the arguments we choose. We have implemented the simulator in such a way that if no arguments are specified, default values are used.

Some of the arguments we can specify are width and height of the mesh, max width and height of a job, which distributions we will use for generating jobs, average job duration, average interval between jobs and number of jobs to create.

7.3 Summary

In this chapter we have described J-Sim and the implementation of our processor allocation simulator. The processor allocation simulator is modular and it is possible to implement any processor allocation strategy into the simulator.

In the next chapter we will discuss the simulation of the previously described processor allocation strategies as a basis for a comparison between these and our Spiral allocation strategy.

Chapter 8

Processor allocation experiments

In this chapter we describe the simulation experiments that were undertaken when studying the processor allocation strategies discussed in Chapter 6. The motivation for this study was to get a basis for a comparison between the discussed processor allocation strategies and our Spiral allocation strategy.

For our simulations we implemented the following processor allocation schemes, First Fit, Best Fit, Adaptive Scan, Random, MC 1×1 and the Multiple Buddy strategy.

We ran the following tests. System utilization analysis of the different allocation schemes, system service time analysis for the different allocation schemes and a processor utilization analysis, that is, an analysis of how the available CPU resources are utilized. The opposite of system utilization is system fragmentation, they are both metrics of how well the different allocation schemes utilizes the available resources.

8.1 System utilization and fragmentation

System utilization and fragmentation measure how well the processor allocation scheme manages to utilize all the processors in the network. System utilization is a metric of how much the processors have been utilized on average during the total simulation time. Fragmentation measures the opposite, how much on average the processors have not been utilized.

The formula for system utilization S is the following [56]:

$$S = \frac{\sum_{1 \leq i \leq w, 1 \leq j \leq h} B_{i,j}}{w * h * T} \tag{8.1}$$

Where w is the width of the mesh and h is the height of the mesh. $B_{i,j}$ is the busy time for the processor in the ith column on the jth row. T is the total running time of the simulation.

The formula for system fragmentation F is:

$$F = 1 - S \tag{8.2}$$

The formula for system load SL is the following [52]:

$$SL = \frac{|R|_{mean} \times JT_{mean}}{w \times h \times IT_{mean}} \tag{8.3}$$

where $|R|_{mean}$ is the average number of processors requested by a job, JT_{mean} is the average running time for a job, IT_{mean} is the average inter arrival time, that is how fast jobs are generated and $w \times h$ is the size of the mesh network.

To see how the different processor allocation strategies perform, we tested the strategies for different loads. To accomplish this we calculated 10 different interval times to use based on Equation 8.3. We kept the maximum job requirement submesh size equal to the size of the mesh. The size of the mesh was 18×18 nodes.

We simulated for loads from 0.1 to 1.0 with 0.1 increments. The different interval times were retrieved by using equation 8.3. Setting the maximum size of a submesh request equal to the size of the mesh leads to $|R|_{mean} = \bar{w} * \bar{h}$ where $\bar{w}$ and $\bar{h}$ are the average size of the width and height of a submesh. When drawing these from a uniform distribution with a range from 1 to 18, $\bar{w}$ and $\bar{h}$ are 9.5, giving us $|R|_{mean} = 90.25$. The job duration, JT_{mean}, is drawn from an exponential distribution with a mean of 100 cycles. The interarrival times are also drawn from an exponential distribution.

We used the following interarrival mean times: 278.55, 139.27, 92.85, 69.64, 55.71, 46.42, 39.79, 34.82, 30.95 and 27.85. The greater the number the lower the load.

To be sure that we simulated long enough we ran each experiment for 10000 jobs. To eliminate the transient periods, we discarded the first 10% and the last 10% of the observations. The system has reached a steady state after the 1000 first jobs.

Figure 8.1 shows the system utilization for the six processor allocation strategies. Of the six different strategies we see that the non-contiguous perform the best. This is no surprise because when allocating in a non-contiguous manner we do not have the problem of external fragmentation, whenever we have enough processors available for an incoming job, we can always allocate. This is not the case for the contiguous allocation strategies.

The Adaptive Scan strategy performs the best of the contiguous strategies. This is because we scan the mesh twice. First we try to allocate a submesh $w \times h$, if this fails we try to allocate a submesh $h \times w$, thus, allocating jobs we would not otherwise have allocated. Best Fit performs a little better than First Fit, this is due to allocating the submesh in the smallest area.

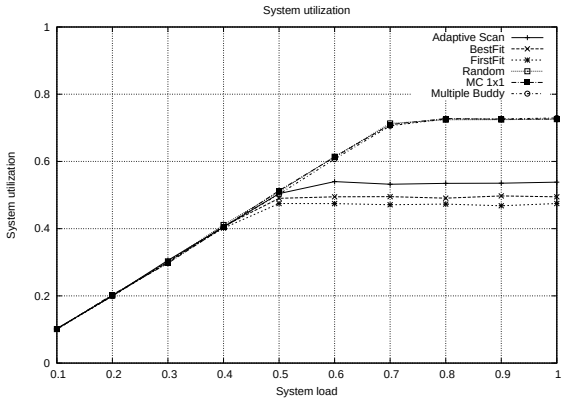


Figure 8.1: System utilization

Strategy	**System utilization**
Adaptive Scan	0.54
First Fit	0.47
Best Fit	0.49
Random	0.72
MC 1×1	0.72
Multiple Buddy	0.72

Table 8.1: System utilization

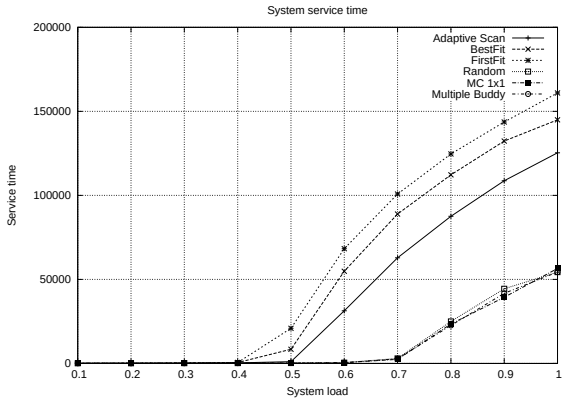


Figure 8.2: System service time

Table 8.1 shows the different processor allocation strategies and their system utilization when the load is 1.

8.2 System service time

A job's service time is a metric of how long a job takes to run, that is the complete time from it is created until it is completed. We see that queuing time is included in the service time. A jobs service time is calculated as the job's running time added to its queuing time.

On average the system service time is:

$$ST = \frac{\sum_{i=1}^{n} QT_i + JT_i}{n} \tag{8.4}$$

where QT_i is the queuing time of the ith job, JT_i is the running time of the ith job and n is the total number of jobs ran.

Figure 8.2 shows the system service time for the 6 processor allocation strategies. We see that the service time for the contiguous allocation strategies increases rapidly after a load of 0.4. The First Fit strategy has the highest service time of the three, Best Fit is second highest and Adaptive Scan is the lowest service time of these allocation strategies, the service time for Adaptive Scan does not start to increase until a load of 0.5.

The system service time for the non-contiguous allocation strategies are kept more constant until a system load of 0.7, then it starts to increase linearly for the loads we experimented with. But the non-contiguous strategies performs better than the contiguous strategies. While the non-contiguous strategies have a system service time of approximately 70000 cycles at a sys-

tem load of 1, the worst of the contiguous strategies, First Fit, have a system service time of approximately 170000 cycles.

If we compare Figures 8.1 and 8.2 we note that there is a correlation. The system utilization for the Best Fit and First Fit strategies starts to flatten at a system load of 0.4, while it is at 0.5 for the Adaptive Scan strategy. These load levels corresponds to where the system service time starts to increase. We see this also for the non-contiguous strategies, their system utilization starts to flatten at a system load of 0.7 which is where the system service time starts to increase.

We have seen that the maximum point for the Best Fit and First Fit strategies to work without contention is a system load of 0.4. After this point, queues will start to build up and jobs require a longer duration to complete. The non-contiguous strategies outperform the contiguous strategies, they can sustain a system load of 0.6 before the queuing of jobs starts to build up and even at a system load of 1 the non-contiguous strategies have much lower system service time than the contiguous strategies. On the other hand, the non-contiguous strategies cause a communication overhead that is not reflected in these graphs.

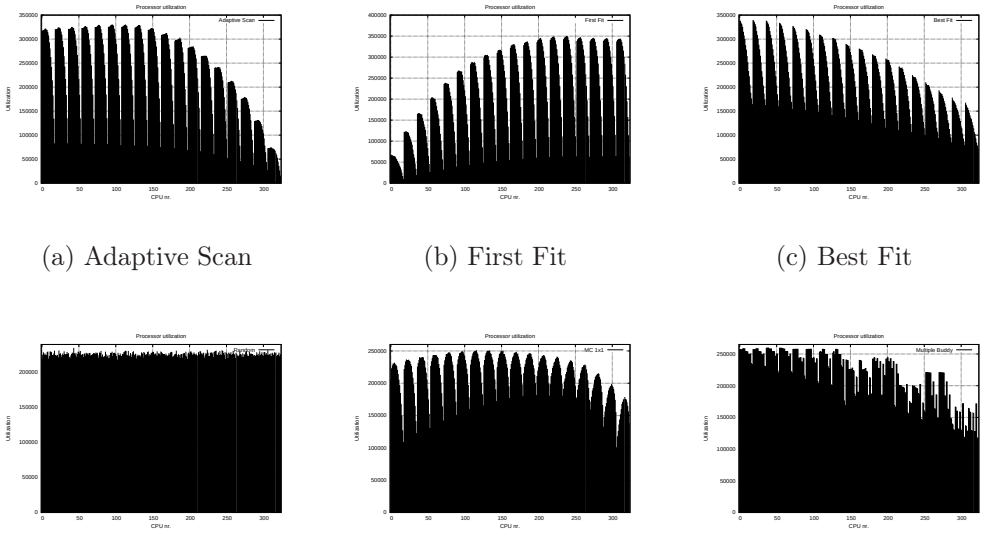
8.3 CPU utilization analysis

In addition to see how the processor allocation strategies performed under different loads, we wanted to perform an analysis of how each processor in mesh was utilized for the different processor allocation strategies. We did this test for each strategy in an 18×18 mesh where the maximum job request submesh size was 18×18 and the system load was 1.

We printed the results from simulations to a file and the numbers were plotted with bars. On the x-axis is the processor ids, the processors are numbered such that processor 0 is the leftmost processor in the first row, processor 18 is the leftmost processor in the second row and so on. On the y-axis is the system utilization time, that is, how much time the corresponding processor has been used during the simulations.

Figure 8.3 shows the individual processor utilization for the six different processor allocation strategies. First we will discuss the contiguous allocation strategies, Adaptive Scan, First Fit and Best Fit. Then we will discuss the non-contiguous strategies, Random, MC 1×1 and the Multiple Buddy strategy.

The Adaptive Scan strategy utilizes more processors in the lower leftmost region of the mesh, while the upper rightmost region is less utilized. This seems reasonable because the scan starts in the lower leftmost corner of the



(a) Adaptive Scan (b) First Fit (c) Best Fit

(d) Random (e) MC 1x1 (f) Multiple Buddy

Figure 8.3: CPU utilization

mesh.

The First Fit strategy utilizes most processors in upper leftmost region, while the lower rightmost region is less utilized, this seems reasonable since the First Fit algorithm scans from left to right and from top to bottom.

The Best Fit strategy has the a similar utilization pattern as Adaptive Scan. Most processors are utilized in the lower leftmost area while the upper rightmost area is less utilized. This seems also reasonable because after we have filled in the coverage matrix, this matrix is scanned from left to right and from bottom to top to find the node with most busy neighbors.

If we compare the Adaptive Scan plot and the Best Fit plot, we see that the processors in Adaptive Scan are more evenly utilized.

As opposed to the contiguous processor allocation strategies, each processor is equally utilized with the Random allocation strategy, which is of no surprise because every processor that is allocated is chosen at random. We see that every processor is utilized for nearly the same period of time.

The MC 1×1 strategy has a pattern similar to the contiguous strategies, but we see that the processor utilization is more evenly distributed across the processors than for the contiguous allocation strategies. This pattern may arise from the fact that we use a scan pattern similar to Adaptive Scan and Best Fit, scanning from each row from the leftmost lower corner. But we allocate available processors in shells around the processor selected as base

for the allocation.

For the Multiple Buddy strategy, processors in the lower leftmost corner of the mesh are most utilized. The utilization decreases slowly while moving towards the upper rightmost region. During the initialization step the algorithm will partition an 18×18 mesh into one 16×16 blocks based in the lower leftmost corner and 17 2×2 blocks. For small jobs, jobs requiring four or less processors, only the smaller blocks will be allocated. For larger jobs, the large block will be partitioned into several smaller blocks until we have the right amount of processors. When we simulated the Multiple Buddy strategy we set the maximum job size equal to the size of the mesh, 18×18. Therefore, it is reasonable to believe that it is the lower leftmost part that will be used the most because the mean size of a job requirement is 9.5×9.5 or 90.25 processors.

We have seen the individual processor utilization for the six processor allocation strategies. The first three were contiguous allocation strategies while the latter three were non-contiguous allocation strategies. The contiguous strategies showed very clear that the processor utilization pattern for each algorithm was dependent on the scan pattern and there was significant difference in how each processor was utilized. The non-contiguous strategies showed some but small patterns in how the allocation strategies work. One thing that was clear from the processor utilization plots was that the non-contiguous allocation strategies utilized the processors more evenly than the contiguous allocation strategies. The Random allocation strategy utilized the processors most evenly, but when it comes to both I/O and job internal communication this algorithm has significant disadvantages because the internal communication of one job will compete with the internal communication of other jobs, this is not the case for the contiguous allocation strategies which are routing contained.

8.4 Summary

In this chapter we have described the simulations we ran to study the processor allocation strategies we have implemented, and the result of our simulations. We noted that the non-contiguous processor allocation strategies performed better than the contiguous strategies with regard to system utilization and system service time. This is because external fragmentation is not a problem for the non-contiguous strategies, as opposed to for the contiguous strategies. We also studied how the different processor allocation strategies utilized the processors in the mesh. The contiguous strategies used that corner of mesh where the search algorithm starts more than the opposite

corner, while the non-contiguous algorithms utilized the processors in the mesh more evenly.

In the next chapter we will discuss I/O communication and perform a study which involves I/O simulation with the different processor allocation strategies. We will then present and study the new Spiral allocation strategy based on Adaptive Scan.

Chapter 9

Job Allocation and I/O performance

In the previous chapter we saw how Adaptive Scan, First Fit, Best Fit, Random, MC 1×1 and the Multiple Buddy strategy performed under different loads with respect to system utilization and system service time. We also saw how each of the mesh connected processors was utilized by the different processor allocation strategies. We did not take communication into account in the evaluation of how the strategies behaved. Intra process communication, message passing between processors that execute a job, is not the only type of communication in a HPC environment. Many of the jobs that run in such systems are typically I/O intensive jobs, which require much disk access, and must therefore also communicate with the I/O nodes.

In this chapter we will discuss I/O communication and introduce a new processor allocation strategy, the Spiral allocation strategy.

9.1 Placement of I/O nodes

In the literature that discusses I/O communication and processor allocation [33, 38], the I/O nodes are placed along one edge of the mesh. According to [38]:

> "The constraint that I/O nodes be located on one side of the mesh derives from pragmatic decisions made by the designers of current supercomputers. These include cost-benefit decisions to exclude I/O functionality from the general compute nodes, as well as cable length, power and heat dissipation considerations."

Both [33] and [38] discuss processor allocation algorithms that reduce I/O communication overhead and how spatial layouts of jobs influence I/O

performance.

The results from the experiments ran by Mache et al. in [38] show that the spatial layout of jobs relative to the I/O nodes affects the network contention level.

In our work we wanted to experiment with a different placement of I/O nodes, instead of placing the I/O nodes along the lower edge of the mesh, the I/O nodes were distributed around the mesh.

9.2 A new processor allocation strategy

We wanted to create a processor allocation strategy that takes the new placement of I/O nodes into account. We believe that a spiral search algorithm will be a good choice in this situation. This is because it will first try to allocate the submesh partitions closest to the edge of the mesh first. We now assume that I/O nodes are placed along every edge of the mesh.

Our new allocation strategy is based on the Adaptive Scan strategy where we have altered the way we scan the mesh. We name our allocation strategy for the *Spiral Allocation* strategy. Our algorithm can be divided into the following three parts.

First we calculate the rejection set and coverage set in the same way as for the Adaptive Scan strategy. The rejection set is calculated from the mesh size and the job size. The rejection set tells us which processors cannot be the base of the allocation because the allocation would partly be placed outside the mesh. The coverage set is calculated from the busy set based on the incoming job size, the busy set tells us which processors that are already allocated. The following example illustrates the information held in the reject set, if we have a submesh of size $w \times h$ and we get a job request of $w' \times h'$ processors arrives, only the processors from $< 0, 0 >$ to $< w - w' + 1, h - h' + 1 >$ can be the base of the allocation.

After we have calculated the coverage set and rejection set, we scan the mesh in a spiral fashion, starting with the lower leftmost node, scanning the mesh clockwise until we find a processor which is not in the coverage set nor in the rejection set. If we find such a processor, this processor will be the base of the allocation and we add the allocated processors to the busy set. If we cannot find such a processor, we then try to allocate the submesh $h' * w'$. If this submesh cannot be found the job is put in the waiting queue. For example, if we find a base located at $< x, y >$ we add the processors from $< x, y >$ to $< x + w' + 1, y + h' + 1 >$ to the busy set. Figure 9.1 shows the spiral search pattern.

The final part is the deallocation, and this is accomplished by removing

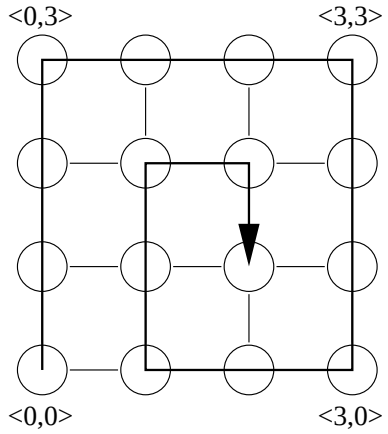


Figure 9.1: The spiral search pattern of the new Spiral Allocation strategy

the processors corresponding to the terminating job from the busy set.

9.3 I/O performance experiments

Before we started experimenting with the new Spiral allocation strategy we experimented with I/O communication for the 6 other processor allocation strategies discussed earlier.

The I/O communication was simulated in the Conan packet simulator, described in Chapter 4 while discussing the LASH experiments. In this section we discuss our implementation to model I/O communication in Conan, the experiments we did and the results from the simulations.

9.3.1 Conan implementations

To simulate I/O communication we had to create 7 new classes. We created a new package called *iosim* which contains *IOSimSetup*, *IOSimRouting*, *IOSimEndNode*, *IOSimIOEndNode*, *IOSimPacket*, *IOSimParameters* and *IOSimTrafficGenerator*.

IOSimSetup contains methods responsible for setting up and starting the simulation. This class contains the main function.

IOSimRouting implements the routing function that is used in our simulation, which is ordinary dimension order routing, XY routing.

IOSimEndNode and *IOSimIOEndNode* implement the end nodes, there are two kinds of end nodes, processor nodes and I/O nodes. The former class implements the processor nodes, the latter the I/O nodes. In our model it is the processor nodes that are the sources of the communication, both intra

job communication and I/O communication, the I/O nodes just receive data from the processor nodes.

IOSimPacket contains packet information.

IOSimParameters contains the extra parameters we want to use. These include which job file and topology file the simulation should use, the width and height of the mesh, how many I/O nodes should be used, where to place the I/O nodes and finally a parameter which tells us how much I/O traffic an end node should generate in percent, ranging from 0 for intra job communication only to 100 for I/O traffic only.

The job file are generated by the processor allocation simulator during simulation, which generates a snapshot of the current situation at some time during simulations. The snapshot tells us which jobs are currently running and which processors are allocated to which jobs. The number of snapshots taken during a simulation is specified when running a simulation.

The topology file contains the topology information for the mesh and is also generated by the processor allocation simulator.

IOSimTrafficGenerator implements the following traffic patterns, uniform I/O traffic, which generates both uniform intra job traffic and I/O traffic where the I/O traffic is distributed uniformly among the I/O nodes. Hot spot I/O traffic, which generates uniform intra job traffic, but every processor node sends I/O traffic to only one randomly pre selected I/O node. Round robin I/O traffic where each processor node generates uniform intra job traffic but generates I/O traffic where the destination I/O nodes are selected in a round robin fashion. Finally, Closest I/O traffic, where each processor sends to the closest I/O node measured in the L_1 distance or *Manhattan distance*, where the distance is measured in total number of vertical hops and horizontal hops from the source end node to the destination end node.

The number of I/O nodes and the placement of the I/O nodes are given as two parameters when running a simulation. The placement of I/O nodes could be either along the lower edge of the mesh or distributed around the mesh. Which switch each I/O node is connected to is stored in an array. To find which processor node to send data to, a record is kept of which processors belong to which jobs. Every node executing a job has an array that contains every other node that belongs to same job. Idle end nodes and I/O nodes do not generate any traffic.

Our model is that intra job traffic is using a uniform traffic pattern and I/O traffic can be one of the patterns described earlier, uniform, hotspot, round robin or closest. When an I/O node receives an I/O packet it does nothing more with that packet, this was done for simplicity. Of course, we could have had the I/O nodes to generate response packets when they received I/O packets. But we believe that this is not that important. This

was an implementation decision that was taken.

9.3.2 Performance analysis

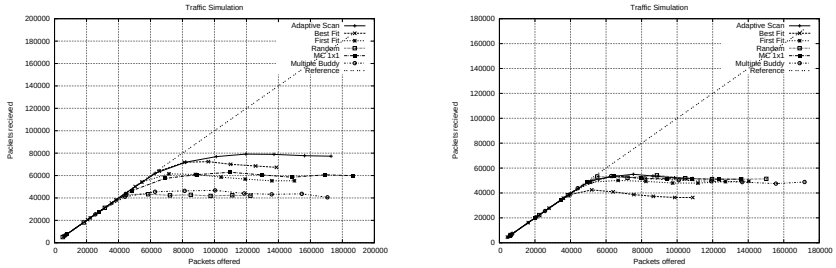
Before we could simulate network traffic for the different processor allocation strategies we have implemented and discussed, we needed to retrieve snapshots from the processor allocation simulator. We ran the simulator for each processor allocation strategy with a system load of 1, meaning full intensity, and retrieved 15 snapshots from the simulation. We then manually checked the snapshots from each allocation strategy to find snapshots that were similar both in number of jobs running and job size. To be sure we retrieved two sets of job files.

Simulation without I/O traffic

Before we tested how the processor allocation strategies performed with I/O traffic we wanted to see how each allocation strategy performed without I/O traffic. This is a test to see how the contiguous allocation strategies performs against the non-contiguous allocation strategies when using intra job traffic only. In [34] Lo et al. shows that contiguous allocation strategies are better than non-contiguous allocation strategies when it comes to job communication. With results from their ProcSimity simulation tool they show that with non-contiguous allocation strategies jobs take longer time to complete because one job's traffic has to compete with other jobs' traffic. The worst allocation strategy is random.

We simulated our mesh network of size 18×18 with two different job sets for each allocation strategy and a range of different traffic loads. The routing function used under simulation was dimension order routing (XY routing). Each switch has one end node. The simulations were run for 30000 cycles.

We see from the results in figure 9.2 that the results for the two different job sets are quite different. For job set 1 we see a trend that the contiguous allocation strategies have a much higher throughput of packets than the non-contiguous allocation strategies have. For job set 2 this trend has disappeared. For this job set it looks like the non-contiguous allocation strategies perform better than First Fit and Best Fit, First Fit performs especially poorly in this simulation, and similar to Adaptive Scan. From these simulations it is clear that the shape of the jobs in the job files used in the simulation are important but it is hard, if not impossible in the model we have chosen to find job files that are equal in both number of allocated jobs and size of jobs. This is because each allocation algorithm gives different results.



(a) Job set 1

(b) Job set 2

Figure 9.2: 18×18 mesh with intra job traffic only

Simulation with I/O traffic

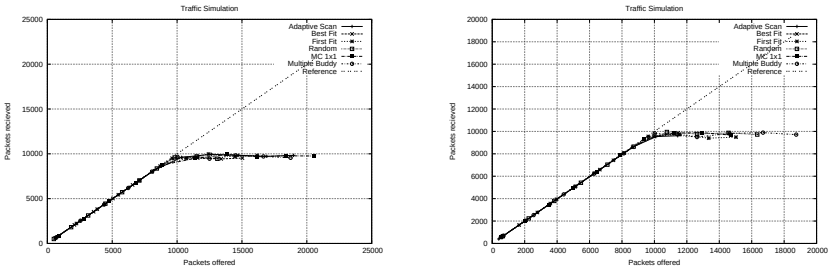
After we simulated pure intra job communication for the jobs allocated by the different processor allocation algorithms, we introduced I/O traffic. Our setup is the following. Our network is still an 18×18 mesh, we test for 4, 8 and 18 I/O nodes, and we test for two levels of I/O load, where 40% and 80% of the traffic is I/O traffic. The traffic pattern we use is unifom I/O traffic.

We have chosen the number of I/O nodes in accordance with previous studies where the number of I/O nodes are equal to or less than the length of one side in the mesh[38, 33], and we do not want to use more I/O nodes than the length of one side in the mesh because we do not want to place I/O nodes along other edges of the mesh in this experiment.

The routing is still dimension order routing, both for job internal traffic and I/O traffic.

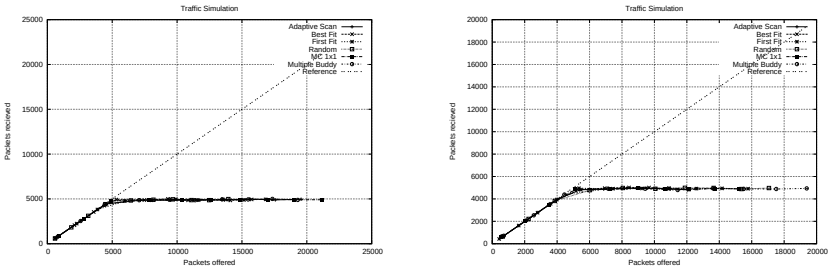
Figure 9.3 and 9.4 show the results for 4 I/O nodes with 40% and 80% I/O traffic respectively. As opposed to the case of pure intra job communication, we see little difference between the two different set of jobs. For 40% of I/O traffic we observe that when the traffic load increases the number of packets that we manage to deliver does not exceed 10000 packets. When we increase the I/O load to 80% the number of delivered packets is reduces to the half of that for 40% I/O traffic, when the traffic increases the number of delivered packets is at maximum approximately 5000. This is the case for both job sets. We also see that there are no significant difference between the different processor allocation strategies when the I/O traffic load increases.

Figure 9.5 and 9.6 show the results for 8 I/O nodes with 40% and 80% I/O traffic respectively. We now see that when doubling the number of I/O nodes the number of delivered packets is also doubled. This is reasonable



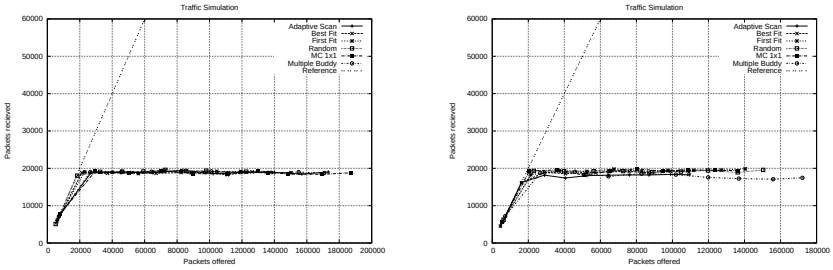
(a) Job set 1 (b) Job set 2

Figure 9.3: 18×18 mesh with 40% I/O traffic with 4 I/O nodes on lower edge



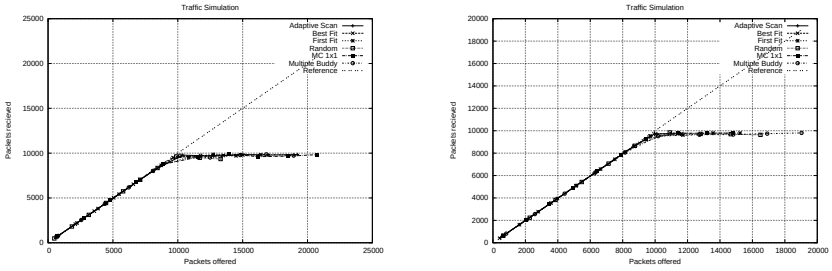
(a) Job set 1 (b) Job set 2

Figure 9.4: 18×18 mesh with 80% I/O traffic with 4 I/O nodes on lower edge



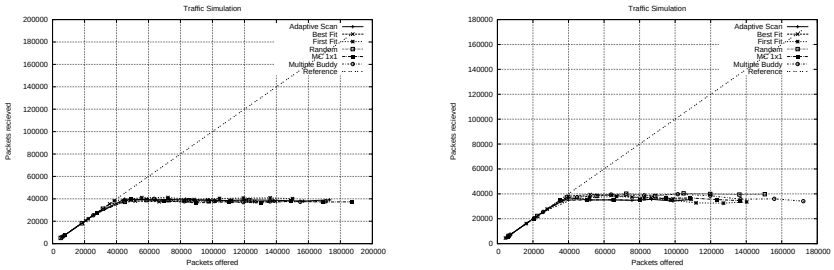
(a) Job set 1 (b) Job set 2

Figure 9.5: 18×18 mesh with 40% I/O traffic with 8 I/O nodes on lower edge



(a) Job set 1 (b) Job set 2

Figure 9.6: 18×18 mesh with 80% I/O traffic with 8 I/O nodes on lower edge



(a) Job set 1 (b) Job set 2

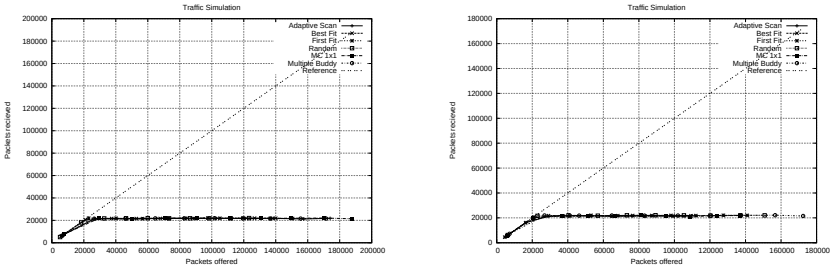
Figure 9.7: 18×18 mesh with 40% I/O traffic with 18 I/O nodes on lower edge

because the 4 I/O nodes used in the previous experiment become a hotspot when I/O traffic increases because the number of I/O nodes is a magnitude lower than the number of processing nodes, which is 324, although some of these are not allocated to any job. Thus, when the number of I/O nodes increases the number of delivered packets increases.

Figure 9.7 and 9.8 show the results for 18 I/O nodes and with 40% and 80% I/O traffic, respectively. Again we see the same pattern as before, and again we see an increase in the maximum number of delivered packets.

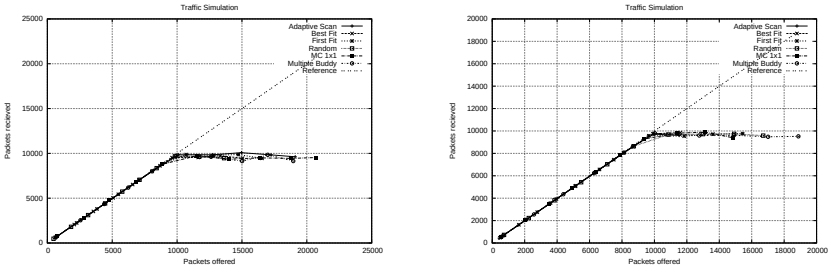
Distributing I/O nodes around the mesh

After we tested throughput for the processor allocation strategies with I/O traffic where the I/O nodes where placed along the lower edge of the mesh,



(a) Job set 1 (b) Job set 2

Figure 9.8: 18×18 mesh with 80% I/O traffic with 18 I/O nodes on lower edge



(a) Job set 1 (b) Job set 2

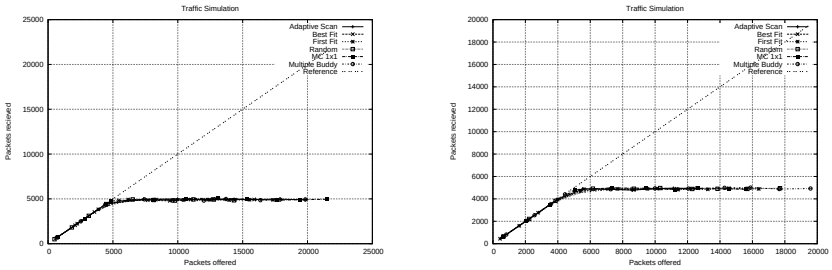
Figure 9.9: 18×18 mesh with 40% I/O traffic with 4 I/O nodes distributed

we wanted to study the impact on throughput when the I/O nodes are distributed along each side of the mesh.

This set of experiments was done the in same way with similar parameter settings as the previous experiments, except that now the I/O nodes are distributed along each side in the mesh. We ran experiments for 4, 8 and 16 I/O nodes, we used these figures because they are divisible by 4, because we want an equal number of I/O nodes along each side of the mesh.

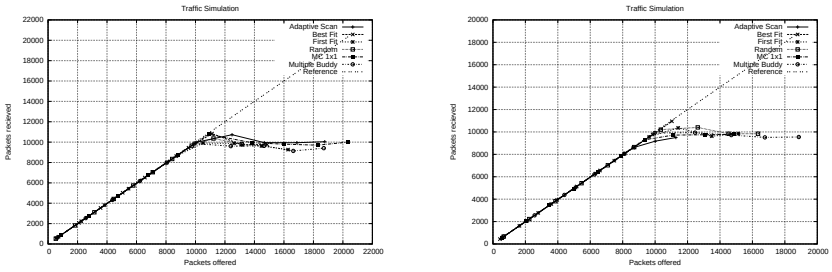
Figures 9.9 and 9.10 show the results for 4 I/O nodes with 40% and 80% I/O traffic respectively. In the results we do not see any difference from the previous experiment with 4 I/O nodes.

Figures 9.11 and 9.12 show the results for 8 I/O nodes with 40% and 80% I/O traffic, respectively. These results show no improvement from using 4 I/O nodes. We wanted to investigate this further, so we add more I/O nodes and test for 16 I/O nodes, that is 4 I/O nodes along each side of the mesh.



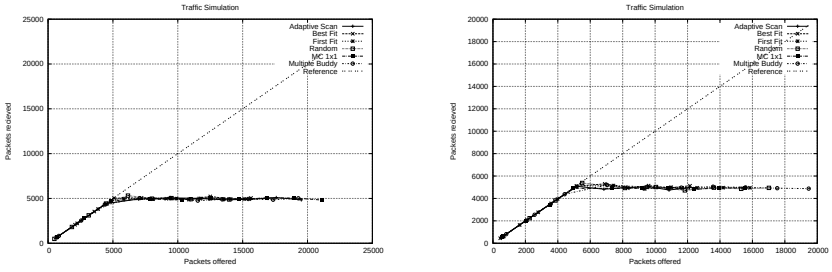
(a) Job set 1 (b) Job set 2

Figure 9.10: 18×18 mesh with 80% I/O traffic with 4 I/O nodes distributed



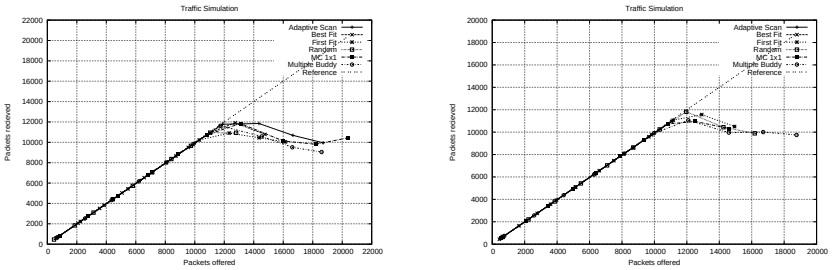
(a) Job set 1 (b) Job set 2

Figure 9.11: 18×18 mesh with 40% I/O traffic with 8 I/O nodes distributed



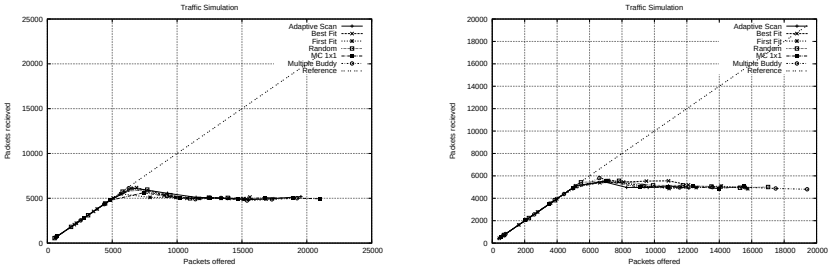
(a) Job set 1 (b) Job set 2

Figure 9.12: 18×18 mesh with 80% I/O traffic with 8 I/O nodes distributed



(a) Job set 1 (b) Job set 2

Figure 9.13: 18×18 mesh with 40% I/O traffic with 16 I/O nodes distributed



(a) Job set 1 (b) Job set 2

Figure 9.14: 18×18 mesh with 80% I/O traffic with 16 I/O nodes distributed

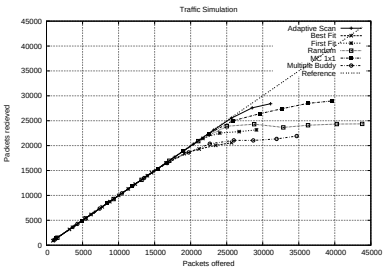


Figure 9.15: 8×8 mesh with intra job traffic only

Figure 9.13 and 9.14 show the result for 16 I/O nodes with 40% and 80% I/O traffic, respectively. Again we do not see much improvement in the number of packets delivered when compared to 4 and 8 I/O nodes. Even though we have 16 I/O nodes, this setting does not perform much better than only having 4 I/O nodes along one side of the mesh. 18 I/O nodes along one side of the mesh gave significant higher throughput. This may indicate that adding more I/O nodes that are distributed around the mesh will not give any improvements. One reason for this may be that there will be much traffic crossing the mesh in different directions when the traffic load is high.

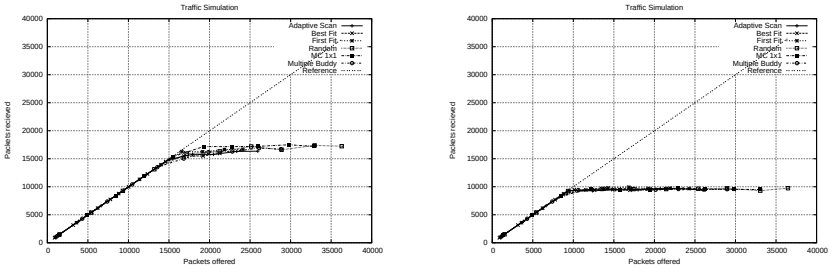
Experimenting with a smaller mesh

A mesh of size 18×18 is a large mesh containing 324 computational nodes, and it is very difficult to find suitable job sets of allocated partitions in a mesh of this size. In addition to the study of a large mesh we wanted to see how the processor allocation strategies performed on a smaller mesh. We therefore ran the same simulations on a mesh of size 8×8. First we ran our processor allocation simulator to retrieve snapshots of the allocated partitions for all our processor allocation algorithms. We retrieved 15 snapshots from every processor allocation strategy and found one suitable job set for each strategy.

Then we simulated for both where I/O nodes were placed along one side of the mesh and where the I/O nodes were placed round the mesh, we only simulated for 8 I/O nodes, equal to the width of the mesh. We simulated for 0%, 40% and 80% I/O traffic. Simulation time was 30000 and we used only 1 end node per switch.

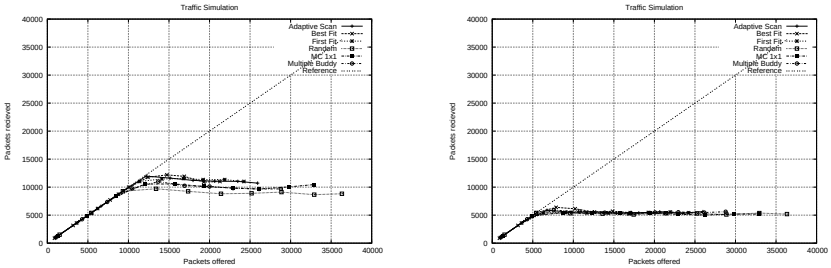
Figure 9.15 displays the result for inta job traffic only, we note that there is a variance between the different processor allocation strategies. This is the same pattern as shown previously.

The figures in Figure 9.16 shows the results for 40% and 80% I/O traffic with I/O nodes placed along the lower edge of the mesh, we note that there is



(a) 40% I/O traffic (b) 80% I/O traffic

Figure 9.16: 8×8 mesh with 8 I/O nodes along lower edge



(a) 40% I/O traffic (b) 80% I/O traffic

Figure 9.17: 8×8 mesh with 8 distributed I/O nodes

now little variance between the different strategies and the number of packets delivered have been reduced from the first simulation using intra job traffic only. This is the same as before.

The figures in Figure 9.16 shows the results for 40% and 80% I/O traffic with I/O nodes distributed around the mesh. We note that the number of delivered packets are much worse than the experiment where the I/O nodes where placed along the lower the edge of the mesh.

9.4 Benchmarking the new strategy

Even though we have shown that distributing the I/O nodes around the mesh may reduce network throughput, we test our new processor allocation algorithm to see how it performs compared to the six other processor allocation strategies, with respect of system utilization, system service time

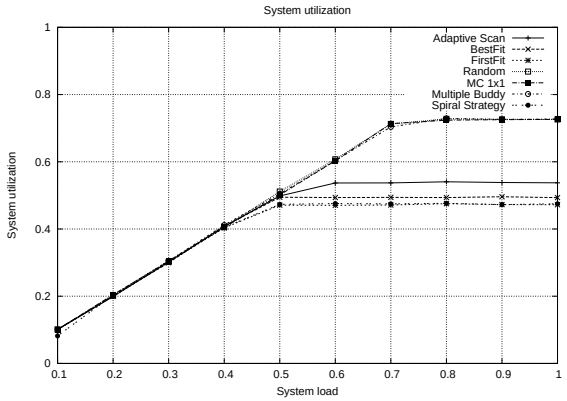


Figure 9.18: System utilization

and individual processor utilization. The spiral strategy is a contiguous processor allocation strategy. Therefore it is most relevant to see how it compares against the other contiguous strategies. We should ensure that our allocation strategy is not worse than First Fit, since this is the strategy that has the worst performance of the three contiguous processor allocation strategies. We cannot think that this algorithm performs similar to the Adaptive Scan strategy since the search pattern has been altered, and this most likely, will affect performance.

The simulation setup was the same as described in Section 8.1. We ran each processor allocation strategy for the same 10 increasing load levels with 5 iterations in each experiment, and calculated the mean of these 5 iterations for each load. Each experiment was run with 10000 jobs and the first 10% and the last 10% observations were removed to remove the transient periods. Again, the only observations of interest are when the system has reached steady state. The maximum job size of each job was equal to the mesh size, 18×18.

Figure 9.18 displays the system utilization for each processor allocation strategy. Except for the added Spiral Allocation strategy this plot is basically the same as Figure 8.1. We note that the system utilization for our Spiral strategy is a little better than the First Fit strategy. This is what we wanted to ensure.

Figure 9.19 displays the system service time for the processor allocation strategies. Again we see that our Spiral Allocation strategy performs a little better than the First Fit strategy.

Figure 9.20 displays the utilization of the individual CPUs in the mesh for the Spiral allocation strategy. On the x-axis is the individual CPUs numbered from 0 to 323. The CPUs are numbered from left to right, from

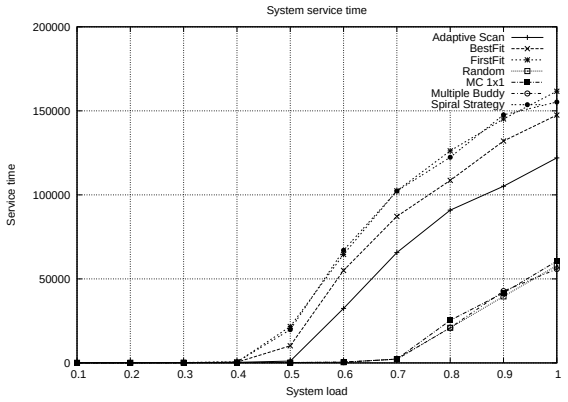


Figure 9.19: System service time

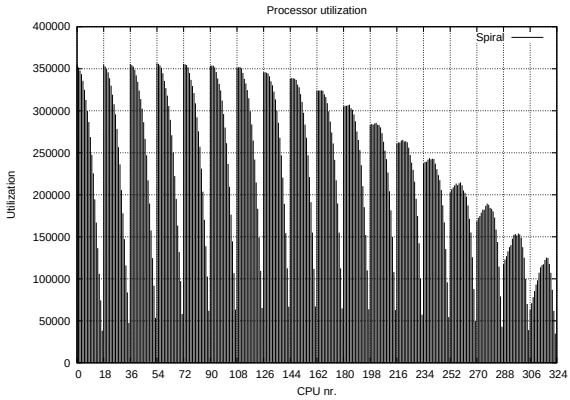


Figure 9.20: Individual processor utilization

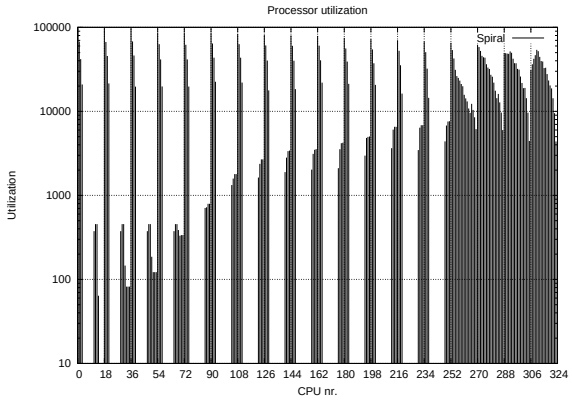


Figure 9.21: Individual processor utilization for small jobs

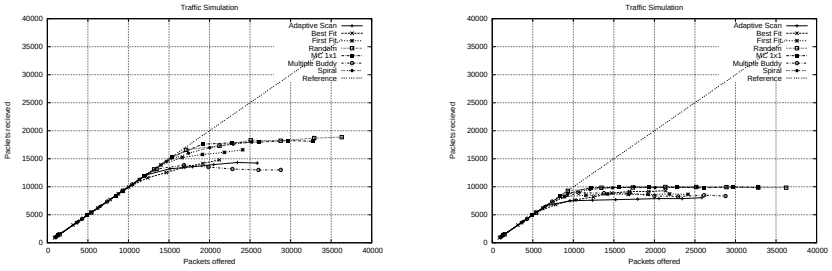
bottom to top. On the y axis is the processor utilization. We see that the pattern is similar to the other contiguous strategies, the lower left area of the mesh is the most utilized. But we see that in the upper part of the mesh, the most used processors are the ones on the right side of the mesh. This is a difference from the other contiguous strategies.

The experiment was repeated one more time, but this time the maximum job size was reduced to 4×4, in other words, very small jobs. The job duration was 100 cycles but the interarrival time between jobs was reduced to 15 cycles to increase the load. The experiment was run with 10000 jobs and the first 10% and the last 10% observations were removed. This experiment was done to see if reducing the job size would have an impact of the individual CPU utilization.

9.21 displays the results with smaller jobs. We see clearly that the edges are more utilized than the center of the mesh, and the upper part of the mesh is more utilized than the lower part of the mesh.

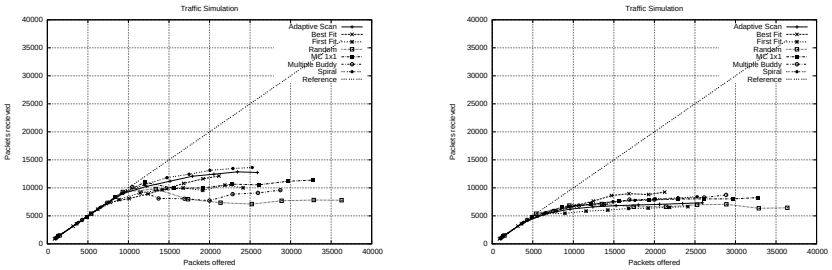
9.5 I/O performance comparison

In this section we repeat the experiments done for an 8×8 mesh but this time the I/O performance for the Spiral processor allocation strategy is included in the experiment. Thus, another traffic pattern was chosen, closest I/O traffic, each processor sends I/O traffic only to the closest I/O node. One normal end node was connected to each switch and 8 I/O nodes were used. The I/O traffic load was 40% and 80% for each strategy. Each simulation was run for 30000 cycles. The job files used are the same as in the previous experiment for First Fit, Best Fit, Adaptive Scan, Random, MC 1×1 and



(a) 40% I/O traffic (b) 80% I/O traffic

Figure 9.22: 8×8 mesh with I/O nodes along one edge



(a) 40% I/O traffic (b) 80% I/O traffic

Figure 9.23: 8×8 mesh with distributed I/O nodes

Multiple Buddy. For the Spiral allocation strategy we had to generate and find a suitable job file. The job file was generated from a simulation in the processor allocation simulator with a system load of 1. 15 job files were generated and the most suiting job file, the one most similar to the other job files, was selected.

Figure 9.22 shows the result from the experiment where the I/O nodes were placed along the lower edge in the mesh. When we compare these results and Figure 9.16, the results varies more between each processor allocation strategy using the closest I/O node than it did when using a random I/O node. But the maximum throughput is approximately the same as the maximum throughput when using random I/O traffic. We note that the Spiral allocation strategy has one of the highest throughput.

For 80% closest I/O traffic, we see the same as before, throughput is reduced for all allocation strategies. In comparison to 80% random I/O traffic, the difference between each strategy is larger but the maximum throughput is

about the same. Again, the Spiral allocation strategy has one of the highest throughput.

Figure 9.23 shows the result when the I/O nodes are distributed around the mesh. Again, when comparing to the results in Figure 9.17, in which random I/O traffic was used, the variation between each processor allocation strategy is greater.

For 40% I/O traffic, we note that the strategy with the highest maximum throughput, the Spiral strategy, has a higher maximum throughput than the processor allocation strategy with highest maximum throughput of packets using uniform I/O traffic. But the knee point in both results are approximately the same. In comparison to the results in Figure 9.22 where I/O nodes are on the lower edge of the mesh, the throughput of packets are generally lower, but the Spiral strategy has approximately equal throughput as the strategy which has the lowest maximum throughput of packets.

For 80% I/O traffic, the results for the distributed I/O nodes shows that the strategies give a little lower throughput than when the I/O nodes are placed on the lower edge, again the Spiral strategy is one of the has one of the highest throughput of packets. In comparison to Figure 9.17, we note that we have higher throughput when using closest I/O traffic than when using random I/O traffic.

In these experiments we have shown that when using closest I/O traffic, sending I/O traffic only to the closest I/O node that is, gives a higher throughput than when using random I/O traffic when the I/O nodes are distributed around the mesh. Our experiments show that our Spiral allocation strategy performs well.

9.6 Further investigation

Because different set of jobs gave different results, another experiment was run to retrieve enough information for statistical comparison. First 22 job sets were gathered during processor allocation simulations for each of the following processor allocation strategies, Adaptive Scan, Best Fit, First Fit, Random, MC 1×1, Multiple Buddy and the Spiral strategy. The experiment was run on an 8×8 mesh and the maximum job size was equal to the mesh. The system load was 1.

Each set of jobs were then simulated in the Conan packet simulator at 10 increasing traffic loads for both 40% I/O traffic and intra job traffic only. 8 I/O nodes were used and the I/O nodes were placed both on the lower edge of the mesh and distributed around the mesh. The traffic pattern were both uniform I/O traffic and closest I/O traffic. For each strategy, the mean of

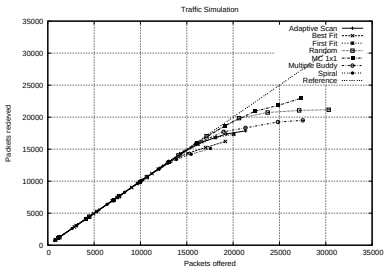
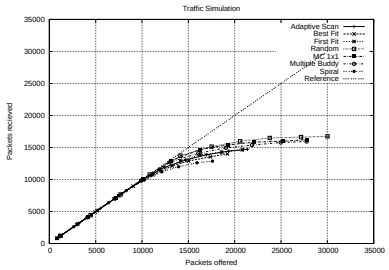
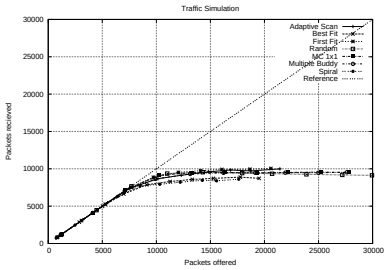


Figure 9.24: Intra job traffic only



(a) I/O nodes along one edge

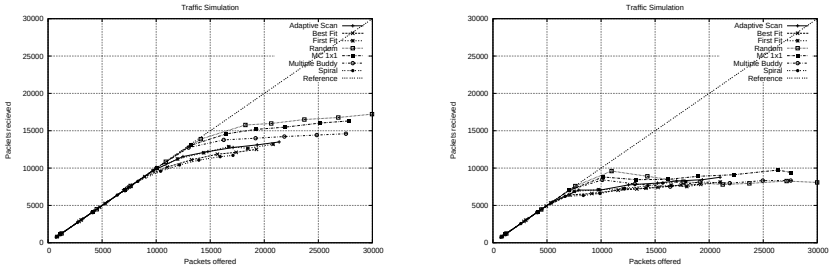


(b) I/O nodes distributed around mesh

Figure 9.25: Uniform I/O traffic

the results for all 22 job sets in each experiment were computed.

Figure 9.24 shows the results for intra job traffic only. Figure 9.25 and Figure 9.26 shows the results for uniform I/O traffic and closest I/O traffic respectively. Again the same trend as previous is shown, when I/O traffic is involved the throughput of packets decreases, and when the I/O nodes are distributed around the mesh, the throughput of packets gets even worse. For some reason when intra job traffic only is used the non-contiguous strategies have the highest throughput of packets, the plot also shows that these strategies generates the most packets, displayed on the x-axis (Packets offered). This may be because the non-contiguous strategies have higher system utilization than the contiguous strategies, thereby having allocated more processors on average and as a result of this, generate more packets. It is possible that this results in higher throughput of packets.



(a) I/O nodes along one edge

(b) I/O nodes distributed around mesh

Figure 9.26: Closest I/O traffic

9.7 Summary

In this chapter we introduced a new processor allocation strategy, the Spiral allocation strategy. This is a contiguous allocation strategy which scans the mesh in a spiral fashion, allocating submesh partitions along the edge of the mesh first. This processor allocation strategy was shown to perform a little better than the First Fit strategy, concerning system utilization and system service time. We also studied I/O performance for the new allocation strategy and the allocation strategies studied in the previous chapter. We showed that when distributing the I/O nodes around the mesh in comparison to having I/O nodes along the lower edge of the mesh, gave significantly lower network throughput, when using random I/O traffic, but when sending only to the closest I/O node, this effect was somewhat reduced.

Using the Spiral allocation strategy and distributing the I/O nodes could be used when the job size is small since we showed that for small jobs, the Spiral strategy would utilize the edges of the mesh more than the center of the mesh.

In the next chapter we will conclude this thesis and discuss our work. The conclusion will end with a discussion of further work.

Chapter 10

Conclusion and future work

This chapter concludes our thesis. After giving an introduction to the research area of interconnection networks, this thesis can be separated into two parts.

1. Implementation and simulation of LASH for the OpenFabrics subnet manager.

2. Implementation and simulation of processor allocation strategies.

In the following sections we will first give a summary of our work with the LASH routing algorithm and the implementation of LASH for the OpenFabrics subnet manager, OpenSM. Then we will give a summary of our work with processor allocation and I/O simulations. After we have given the summaries of the work we will give a critique of the work and present what can be done for future work.

10.1 LASH and OpenFabrics

In the first part of this thesis, we studied the LASH routing algorithm, optimized it and implemented it for the OpenFabrics Subnet Manager (OpenSM). With the implementation of LASH for the OpenSM, we have brought a former theoretical routing algorithm to the HPC market. Leading companies in the HPC industry, e.g. SGI, have become interested in this technology, and have started using this technology in their products.

First we studied the original LASH routing algorithm, to see how it performed with three different topologies, rings, mesh and 2D tori. With performance we mean how many virtual layers we need for LASH to give us a deadlock free shortest path routing. What we noted during these simulation

was that for 2D torus networks, the number of virtual layers needed grew quickly as the network size grew. In addition LASH was slow, for large networks the route calculation time grew exponentially. Memory usage was also high.

We then optimized LASH by modifying the shortest path algorithm, implementing a breadth first search algorithm instead of the shortest path algorithm used in the first version. Testing the new algorithm, we noted that for 2D torus networks the virtual layers needed were now constant and route calculation time and memory usage decreased. The motivation for optimizing LASH was to reduce the number of virtual layers needed to make it scalable for use with InfiniBand which hardware, even though the specification says it supports 16 virtual layers, does not support more than 8, most hardware only support 4 virtual layers.

We then implemented LASH-RP, a variant of LASH where we have the constraint that a source and destination pair path and the opposite source and destination pair path have to be placed in the same virtual layer. The motivation behind this was that the upper layer protocols assume that both paths are using the same virtual layer. We simulated this and showed that it did not perform well on 2D torus and ring networks, and we also proposed a solution to improve performance.

We also tested the variants of LASH with link failures and noted that the optimized version of LASH performed best of the four variants, and LASH-RP performed worst.

We also ran packet simulations. From these results we noted that our optimized version of LASH performed much better than the original version. Also, surprisingly, LASH-RP performed better than all the variants of LASH. This was because LASH-RP distributes source and destinations pairs more evenly across the virtual layers. With performance, in this case, we mean the number of packets delivered when traffic load grows.

When our simulations were complete, we implemented LASH as a routing alternative into the OpenFabrics subnet manager, OpenSM. OpenFabrics is an open source implementation of the InfiniBand network stack. LASH is now available as a routing function using virtual layers in OpenSM with OFED 1.3.

10.2 Processor allocation

In the second part of this thesis we studied processor allocation strategies for mesh topologies and proposed and studied a new processor allocation strategy for the mesh topology, the Spiral allocation strategy. This strategy

tries to allocate submesh partitions along the edge of the mesh first. The motivation behind this strategy was to allocate partitions as close to the I/O nodes as possible. This allocation strategy assumes that the I/O nodes are placed around the mesh and not as in the ordinary manner, along one edge of the mesh.

First we studied six different processor allocation strategies, First Fit, Best Fit, Adaptive Scan, Random, MC 1×1 and Multiple Buddy. The first three are contiguous processor allocation strategies, while the last three are non-contiguous strategies. We showed that non-contiguous allocation performed better than the contiguous strategies with respect of system utilization and system service time. The downside with these allocation strategies is that they cause a communication overhead. We also studied how the different allocation strategies utilize the individual processors in the mesh. It was shown that the non-contiguous strategies utilized the processors more evenly than the contiguous strategies. The contiguous strategies utilized the processors in the corner of the mesh where the search for a free submesh starts, more than in the opposite corner.

These six processor allocation strategies were later used in a I/O communication study, where we used snapshots from the processor allocation simulator we developed, to simulate packet traffic for intra job and I/O communication. A snapshot is a picture of the mesh at some time during processor allocation simulations. It was shown that, even though the results for different snapshots varied, the contiguous allocation strategies gave higher throughput of packets than the non-contiguous strategies. When I/O traffic increased, the difference between the allocation strategies decreased.

The Spiral allocation strategy which was proposed is a contiguous processor allocation strategy which is based on the Adaptive Scan strategy but has a different search pattern. Instead of searching the mesh from left to right, from top to bottom, the strategy searches the mesh in a spiral fashion for a free submesh. The idea is to allocate submeshes as close as possible to the edge of the mesh, and utilize I/O nodes that are distributed around the mesh. In experiments the Spiral allocation strategy was shown to perform equal to First Fit with respect of system utilization and system service time. It was also confirmed that when the job size was small, the Spiral allocation strategy would utilize the edges of the mesh more than the center of the mesh.

During I/O communication studies, it was shown that distributing the nodes around the mesh reduced throughput of packets when using a random I/O traffic pattern. Using the closest I/O node also gave a reduction in throughput of packets, but not as high as random I/O traffic. The spiral strategy was shown to perform well during packet simulations, with regard

to throughput compared to the other processor allocation strategies.

10.3 Critique of work

The LASH routing algorithm has been shown to work in InfiniBand networks, but only for network topologies that require only one virtual layer, e.g. meshes where the LASH algorithm can be reduced to Dimension Order Routing. When the implementation of LASH was done, we did not have access to InfiniBand hardware, we only had access to an InfiniBand network simulator to test if the route calculation worked correctly. We could not test for network traffic, or if the network would deadlock. We discussed in Chapter 4 a strategy for selecting Virtual Lanes (VL) using the PathRecord query. This strategy for selecting VL's has not been fully tested and could be wrong.

We presented a new simulation strategy, the Spiral allocation strategy which uses a spiral search pattern to allocate submeshes first along the edge of the mesh to utilize I/O nodes that are distributed around the mesh. Other works that have discussed processor allocation and I/O communication [33, 38] agree that the I/O nodes should be placed along one edge in the submesh. During the studies, it was shown that distributing the I/O nodes around the mesh gave a significantly reduced throughput of packets. We therefore have to rethink this strategy.

10.4 Future work

We round up this thesis with suggestions to future work based on the work we have done and described in this thesis.

In the first part of this thesis we studied, optimized and implemented the LASH routing algorithm for the OpenSM. Using a simulator tool, it was shown that LASH worked correctly with OpenSM, and industry partners showed that LASH worked on InfiniBand hardware on topologies requiring one data VL. If, in the future, InfiniBand hardware were provided, future work could be to test out LASH on topologies requiring more than one data VL, and study the use of VLs in InfiniBand.

In the second part of this thesis we studied processor allocation strategies and introduced the Spiral allocation strategy for allocating processors as close as possible to the edge of the mesh when I/O nodes are distributed around the mesh. It was shown that distributing I/O nodes around the mesh actually gave a performance reduction, compared to having the I/O nodes

along one edge of the mesh. Our I/O study used snapshots of allocations for packet simulation. This might not be the best model, but because the processor allocation simulator and the packet simulator uses two completely different time scales, we chose to use snapshots. For future work, it could be possible to extend the processor allocation simulator to take I/O into account, simulating both processor allocation and packet traffic at the same time. We did neither experiment with I/O traffic for small jobs. This could also be a part of the future work.

Appendix A

Acronyms

Below is a list of acronyms and their meaning that have been used in this thesis.

- BTH - Base Transport Header
- CA - Channel Adapter
- HCA - Host Channel Adapter
- TCA - Target Channel Adapter
- GID - Global ID
- DGID - Destination Global ID
- SGID - Source Global ID
- GRH - Global Routing Header
- GUID - Global Unique ID
- IBA - InfiniBand Architecture
- IOU - I/O Unit
- LASH - LAyered SHortest path
- LID - Local ID
- SLID - Source LID
- DLID - Destination LID

- LRH - Local Routing Header
- PSN - Packet Sequence Number
- QP - Queue Pair
- RDMA - Remote Direct Memory Access
- SL - Service Level
- SM - Subnet Manager
- VL - Virtual Lane

Bibliography

[1] Sun java technology. http://java.sun.com.

[2] Ieee 802.3 ethernet. http://www.ieee802.org/3.

[3] Openfabrics alliance. http://www.openfabrics.org.

[4] Computational plant. http://www.cs.sandia.gov/cplant.

[5] Commons-math. http://commons.apache.org/math.

[6] J-sim homepage. http://www.j-sim.org.

[7] Top500 supercomputing sites. http://www.top500.org.

[8] Procsimity. http://www.cs.uoregon.edu/research/DistributedComputing/ProcSimity.html%.

[9] Earth simulator. http://www.jamstec.go.jp/es/en/index.html.

[10] Mellanox. http://www.mellanox.com.

[11] Pci-sig. http://www.pcisig.com.

[12] Fcia. http://www.fibrechannel.org.

[13] Y. Aridor, T. Domany, O. Goldshmidt, J. E. Moreira, and E. Shmueli. Resource allocation and utilization in the blue gene/l supercomputer. *IBM Journal of Research and Development*, 49(2-3):425–436, 2005.

[14] InfiniBand™ Trade Association. Infiniband™architecture specification volume 1, release 1.2, 2004.

[15] M. A. Bender, D. P. Bunde, E. D. Demaine, S. P. Fekete, V. J. Leung, H. Meijer, and C. A. Phillips. Communication-aware processor allocation for supercomputers: Finding point sets of small average distance. *Algorithmica*, 50(2):279–298, 2008.

[16] D. P. Bunde, V. J. Leung, and J. Mache. Communication patterns and allocation strategies. *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 15:248b, 2004.

[17] A. A. Chien and J. H. Kim. Planar-adaptive routing: low-cost adaptive networks for multiprocessors. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 268–277. ACM, 1992.

[18] H. Choo, S. M. Yoo, and H. Y. Youn. Processor scheduling and allocation for 3d torus multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):475–484, 2000.

[19] P. J. Chuang and N. F. Tzeng. An efficient submesh allocation strategy for mesh computer systems. *11th International Conference on Distributed Computing Systems*, pages 256–263, 1991.

[20] P. J. Chuang and C. M. Wu. An efficient recognition-complete processor allocation strategy for k-ary n-cube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 11(5):485–490, 2000.

[21] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2004.

[22] J. Ding and L. N. Bhuyan. An adaptive submesh allocation strategy for two-dimensional mesh connected systems. In *ICPP*, pages 193–200, 1993.

[23] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., 2002.

[24] M. Feldman. Hpc in the land of 24/7. `http://www.hpcwire.com/hpc/1906060.html`, 2007.

[25] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 278–287. ACM, 1992.

[26] V. Gupta and A. Jayendran. A flexible processor allocation strategy for mesh connected parallel systems. *icpp*, 03:166–173, 1996.

[27] M. Hussain, R. Gupta, and T. Liu. Using openfabrics infiniband for hpc clusters. *Dell Power Solutions*, 2006.

[28] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[29] M. Kang, C. Yu, H. Y. Youn, B. Lee, and M. Kim. Isomorphic strategy for processor allocation in k-ary n-cube systems. *IEEE Transactions on Computers*, 52(5):645–657, 2003.

[30] M. Koibuchi, A. Jouraku, H. Amano, and A. Funahashi. L-turn routing: An adaptive routing in irregular networks. *ICPP*, 00:383–392, 2001.

[31] S. O. Krumke, M. V. Marathe, H. Noltemeier, V. Radhakrishnan, S. S. Ravi, and D. J. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 181(2):379–404, 1997.

[32] K. Li and K. H. Cheng. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, pages 22–27. ACM Press, 1990.

[33] P. Liu, C. C. Hsu, and J. J. Wu. I/o processor allocation for mesh cluster computers. *Proceedings. 11th International Conference on Parallel and Distributed Systems, 2005.*, 1:105–111, 20-22 July 2005.

[34] V. Lo, K. J. Windisch, W. Liu, and B. Nitzberg. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, 1997.

[35] O. Lysne, S. A. Reinemo, T. Skeie, Å. G. Solheim, T. Sødring, L. P. Huse, and B. D. Johnsen. The interconnection network - architectural challenges for utility computing data centres. *IEEE Computer*, 2007.

[36] O. Lysne and T. Skeie. Load balancing of irregular system area networks through multiple roots. In *Proceedings of the International Conference on Communication in Computing, CIC 2001*, pages 165–171. CSREA Press, 2001.

[37] O. Lysne, T. Skeie, S. A. Reinemo, and I. Theiss. Layered routing in irregular networks. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):51–65, 2006.

[38] J. Mache, V. Lo, and S. Garg. The impact of spatial layout of jobs on i/o hotspots in mesh networks. *Journal of Parallel and Distributed Computing*, 65(10):1190–1203, 2005.

[39] J. Mache, V. Lo, and K. Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing Systems*, pages 120–124, 1997.

[40] W. Mao, J. Chen, and W. III Watson. Efficient subtorus processor allocation in a multi-dimensional torus. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 53. IEEE Computer Society, 2005.

[41] K. Pawlikowski. Steady-state simulation of queueing processes: survey of problems and solutions. *ACM Computing Surveys*, 22(2):123–170, 1990.

[42] W. Qiao and L. M. Ni. Efficient processor allocation for 3D Tori. In *Proceedings of International Parallel Processing Symposium*, pages 466–471, 1995.

[43] W. Qiao and L. M. Ni. Adaptive routing in irregular networks using cut-through switches. In *ICPP, Vol. 1*, pages 52–60, 1996.

[44] R. J. Recio. Server i/o networks past, present, and future. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 163–178. ACM Press, 2003.

[45] J. C. Sancho, A. Robles, and J. Duato. An effective methodology to improve the performance of the up*/down* routing algorithm. *Transactions on Parallel and Distributed Systems*, 15(8):740–754, 2004.

[46] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318–1335, Oct 1991.

[47] K. H. Seo. Fragmentation-efficient node allocation algorithm in 2d mesh-connected systems. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, pages 318–323. IEEE Computer Society, 2005.

[48] T. Shanley. *InfiniBand Network Architecture*. MindShare, Inc., 2003.

[49] F. Silla and J. Duato. On the use of virtual channels in networks of workstations with irregular topology. *IEEE Transactions on Parallel and Distributed Systems*, 11(8):813–828, 2000.

[50] T. Skeie, O. Lysne, J. Flich, P. Lopez, A. Robles, and J. Duato. Lashtor: A generic transition-oriented routing algorithm. In *ICPADS '04: Proceedings of the IEEE International Conference on Parallel and Distributed Systems*, pages 595–604. IEEE Computer Society, 2004.

[51] T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (lash) routing in irregular system area networks. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 194. IEEE Computer Society, 2002.

[52] Å. G. Solheim, O. Lysne, T. Sødring, T. Skeie, and J. A. Libak. Routing-contained virtualization based on up*/down* forwarding. In *High Performance Computing - HiPC 2007*, pages 500–513. Springer-Verlag, 2007.

[53] I. Theiss. *Modularity, Routing and Fault Tolerance in Interconnection Networks*. PhD thesis, University of Oslo, 2004.

[54] K. Windisch, V. Lo, and B. Bose. Contiguous and non-contiguous processor allocation algorithms for k-ary n-cubes. In *Proceedings of the 24th International Conference on Parallel Processing*, pages II:164–168, 1995.

[55] F. Wu, C. C. Hsu, and L. P. Chou. Processor allocation in the mesh multiprocessors using the leapfrog method. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):276–289, 2003.

[56] Y. Zhu. Efficient processor allocation strategies for mesh-connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328–337, 1992.